Visualising Topological Structures of Activation in Artificial Neural Networks

Terence Broad

Supervisor: Mick Grierson

Abstract

This project report describes a first approach to creating a visualisation of an artificial neural network, that visualises the topology of the network given an individual data input that the network has learned to recognise. A survey of previous attempts to visualise both artificial and biological neural networks is presented, as well as a survey of various techniques used in other forms of network visualisation that could be applied to visualising artificial neural networks. This is followed by a detailed description of the method implemented in this project, followed by results from the visualisation.

1 Introduction

In recent years a handful of techniques (under the umbrella of deep learning) have quickly superceded most other state of the art techniques in machine learning at a wide variety of tasks: image recognition, speech recognition, machine translation, natural language processing, etc. At the core of all these algorithms is some form of artificial neural network, networks of interconnected nodes with adaptive weights assigned to each connection. These weights are adapted by a learning rule during training until the network converges on a set of suitable weights that maximises (or minimises) the given training heuristic. These algorithms generalize very effectively and can be used with massively varying kinds of datasets without any need for hand engineering of features.

In spite of their effectiveness, neural networks can be notoriously difficult to comprehend, many modern neural network have many thousands of nodes and millions of connections between nodes. In addition to this modern neural network frameworks are seen as black-box architectures, models are defined as hierarchies of tensors and some hyperparameters can be tuned, but beyond this there is very little in the way of accessible and understandable principles underlying how a neural network has converged on a solution to a problem. Some excellent efforts have been

made in order to visualise maximal activations of units in convolutional neural networks used for image recognition, and the hierarchy of increasingly complex features that have been activated contributing to the high level object recognition task.

What has not been done effectively is visualisations of the emergent topological structures that present themselves after training and through exposure to data structures that the network has learnt to recognise. One of the obvious difficulties in doing this is the size and scale of these modern artificial neural networks, however alongside recent developments in neural networks has developed the discipline of large scale data visualisations, and more specifically, large scale network visualisations. Thus the goal of this project is to apply appropriate methods of large scale network visualisations toward exposing the inner workings of some of the most successful and well known neural network architectures.

2 Context

In this chapter a short overview of the basic principles of neural networks is given in section 2.1, as well as detailed explanations of the network architectures that are to be visualised later in the project. Descriptions and examples of different network visualisation techniques is detailed in section 2.2. To conclude the chapter a survey of existing attempts to visualise biological neural networks (2.3) and artificial neural networks (2.4) is given.

2.1 Artificial Neural Networks

Artificial neural networks are a family of statistical models that are inspired by biological neural networks, specifically the central nervous system of animals (i.e. the human brain). Artificial 'neurons' or nodes are connected together to form a network that somewhat mimics a biological neural network. Each node in the network takes a weighted sum of its inputs, and often performs some form of nonlinear function on this some (often referred to as the activation function). The numerical weights assigned to these connections are updated and tuned according to some kind of learning rule, often through some method of backpropagating errors derivatives back through the network. Through exposure to data, these models can develop approximations of nonlinear functions of high dimensional inputs.

2.1.1 Convolutional Neural Networks

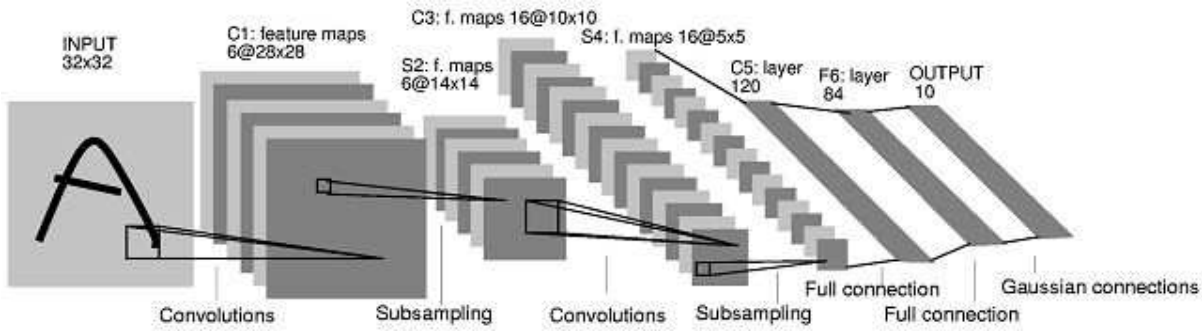

Fig.2.1.1 A diagram of the architecture of LeNet-5, one of the earliest examples of a convolutional neural network used for image recognition. [LeCun et al. 1998]

Convolutional neural networks are one of the most successful artificial neural network models to have been directly inspired by biological processes. Mimicking the hierarchy of overlapping filters and regions present in the human visual system [Cichy et al. 2016]. What differentiates convolutional neural networks from standard multilayered perceptrons is the sparsely connected convolutional layers, where filters are learned using overlapping sets of shared weights. Different configurations of

shared weights correspond to and are constrained to different feature maps. These filter maps are are reduced in size often using average-pooling or max-pooling functions (where the node of strongest activation is chosen from regular regions of the filter maps). Layers of convolutions followed by max-pooling functions are arranged in a hierarchical order until these maps becomes very small, where they are usually followed by several fully connected layers and then a softmax layer.

Early examples of convolutional neural networks date back to the work of Fukushima [1980], but their design was improved and proven to be useful practically by LeCun et al. [1998] with the network architecture known as LeNet-5 (see Fig.2.1.1) that was used for many years for automatic recognition of digits in bank checks. Alex Krizhevsky et al. [2012] massively outperformed the state of the art in image recognition methods using a very deep convolutional neural network (which had 60 million parameters and 650,000 neurons) in the ImageNet competition. Making obsolete methods of laboriously hand engineered features combined with more traditional machine learning methods and propelling convolutional neural networks to the forefront of research in machine learning.

2.1.2 Recurrent Neural Networks

Recurrent neural networks are a class of artificial neural network where connections between units produce directed cycles, as opposed to purely feed forward networks such as multilayered perceptions or convolutional neural network. Having directed cycles allows the network to develop internal states that can exhibit dynamic temporal behaviour.

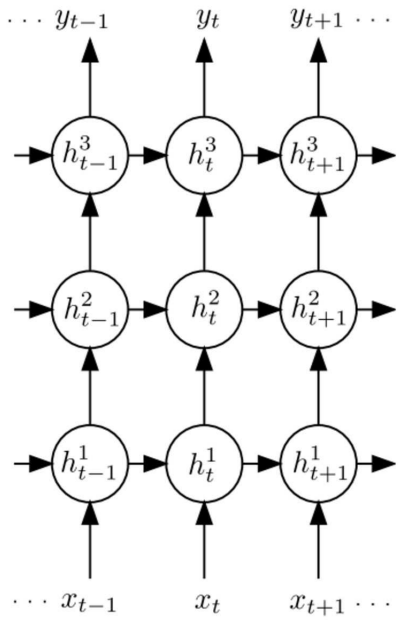

Fig.2.1.2 A diagram of a deep recurrent neural network unrolling through discrete time steps. [Graves et al. 2013]

There are many different types of recurrent neural network architectures, such as fully recurrent networks, hopfield networks and echo state networks. However despite showing early promise in the 1980's, progress in the practical application of these networks was hampered as recurrent neural networks were particularly susceptible to the vanishing gradient problem [Hochreiter et al. 2001]. It was not until the development of Long Short Term Memory Networks (LSTM) that a robust solution to this problem developed.

2.1.2.1 Long Short Term Memory Networks

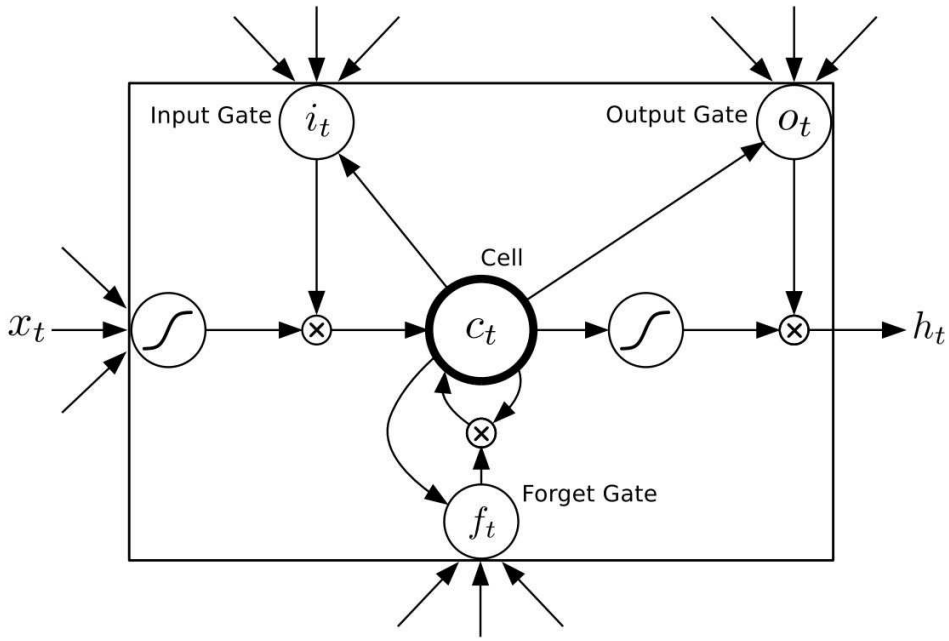

Fig.2.1.3 A diagram of a Long Short Term Memory network cell block. [Hochreiter and Schmidhuber 1997]

Long Short Term Memory Networks (LSTM) are a recurrent neural network architecture developed to address the vanishing gradient problem and to process, classify and predict in time series where long periods of time of unknown lengths exist between important events, they were developed by Sepp Hochreiter and Jürgen Schmidhuber in [1997]. In 2009 LSTM networks achieved the best known results in connected handwriting recognition in many languages with no prior knowledge of the given languages [Graves et al. 2009] and in 2013 were successfully used in automatic speech recognition [Graves et al. 2013]. LSTM networks have proven to be powerful and often amusing generative models for producing sequences. Alex Graves [2013] demonstrated that LSTM networks could be used to generate sequences of text (and other data structures) by training the network to sequentially predict the next character and training it on large datasets of text (such as wikipedia), once the networks are trained they are then fed back the characters they predict and begin to hallucinate large sequences [Graves 2015]. Below is an example of such a sequence generated by an LSTM network trained on the whole body of Shakespeare's work:

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour. [Karpathy 2015]

2.2 Network Visualisation Techniques

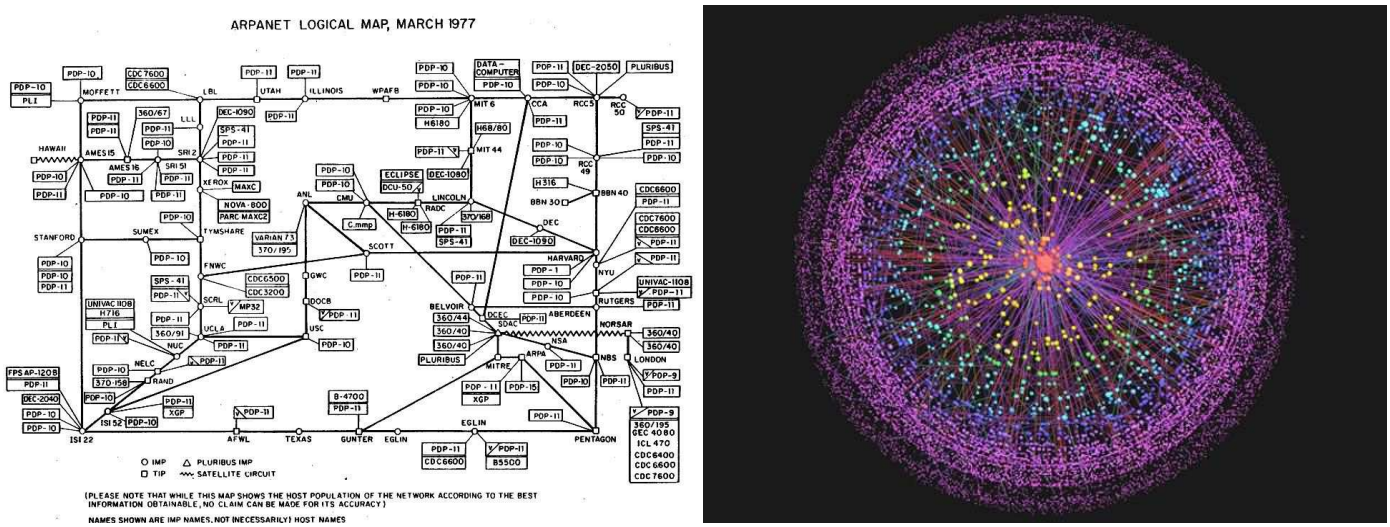

Fig.2.2.1 (Left) A visualisation of the whole of ARPANET in 1977. [Heart et al. 1978]
Fig.2.2.2 (Right) A visualisation showing the hierarchical structure of the internet. [Carmi et al. 2007]

The most common form of network visualisation is node-link diagrams, in fact all of the examples in this section are various forms of node-link diagrams. There are alternatives methods of visualising networks such as intersection graphs [Erdos 1966], circle parkings [Koebel 1936] or adjacency matrices, but node link diagrams are by far the most commonly used and most intuitive to understand. Vertices (nodes) in the network are usually represented as boxes, circles or by textual information, and edges are represented as line segments. Origins of node-link diagrams can be traced back to the 13th Century work of Ramon Llull [Knuth 2013] who was the first to draw diagrams of complete graphs.

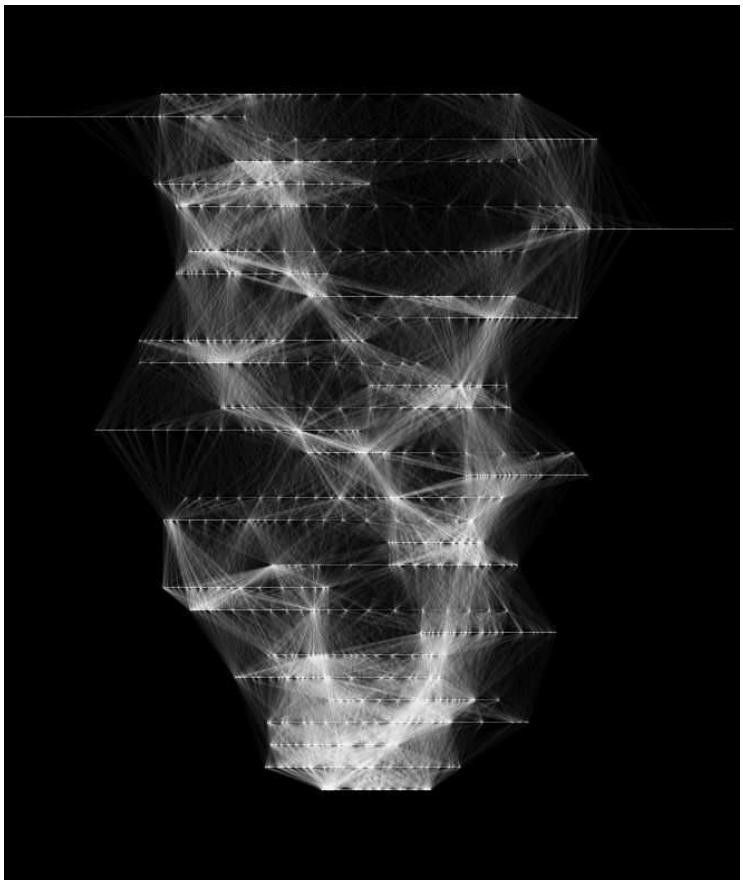

Fig.2.2.3 Sound Sculptures: Custom software for visualizing patterns from sound data in real time. [Broyez 2013]

In graph theory, a graph is defined as a representation of a set of objects (often referred to as nodes or vertices) where pairs of objects are connects by links (often referred to as edges). When graphs are represented visually, the layout of where nodes are positioned in reference to each other has no effect on the structure of the graph itself, but it does have an effect on the ability for people viewing the graph to understand the topology of the graph and to draw meaning from a given visualisation. Thus much of the work of network visualisation goes into finding a suitable method for determining the layout of a network. Nodes can be laid out according to predefined geometric structures such as lines (2.2.2), grids (Fig.2.2.3), circles (2.2.3) or in reference to geographical position (Fig 2.2.4). Another approach is to determine the layout using a force-directed graph drawing algorithm, in which the position of nodes are determined by forces acting upon their relative positions to minimise the number of edges crossing and regularise the length of the edges [Kobourov 2012], an example of this can be seen in Fig.2.2.2. The following subsections describe a selection of layout strategies that may be useful for this project.

Fig.2.2.4 A visualisation of all facebook friend connections mapped to a globe. [Butler 2010]

2.2.1 Trees

In graph theory, a tree is a special kind of graph in which any two vertices are connected by exactly one path, and there are no closed loops or *cycles*. They are an important data structure that is widely used in computer science. In addition to this tree diagrams are some of the oldest and most widely used forms of visualisation [Lima 2011, 2014]. Traditionally used to map hierarchies of knowledge, or genealogies of individuals (family trees).

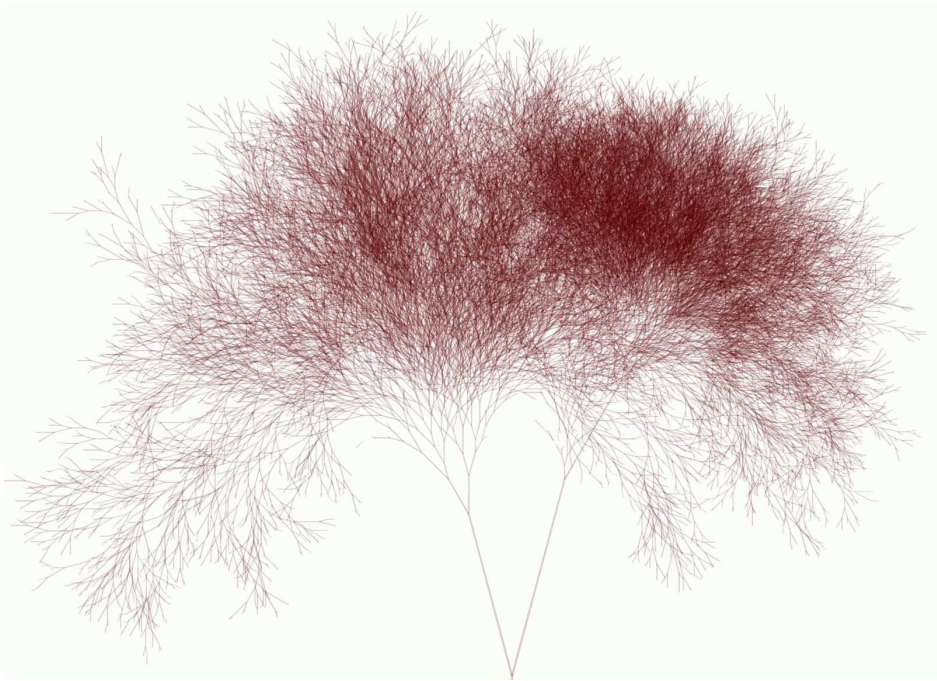

Fig.2.2.5 A visualisation of a binary search tree of 2,147,483,647 indexed web pages by a Yahoo! search bot. [Riet 2006]

2.2.2 Arc Diagrams

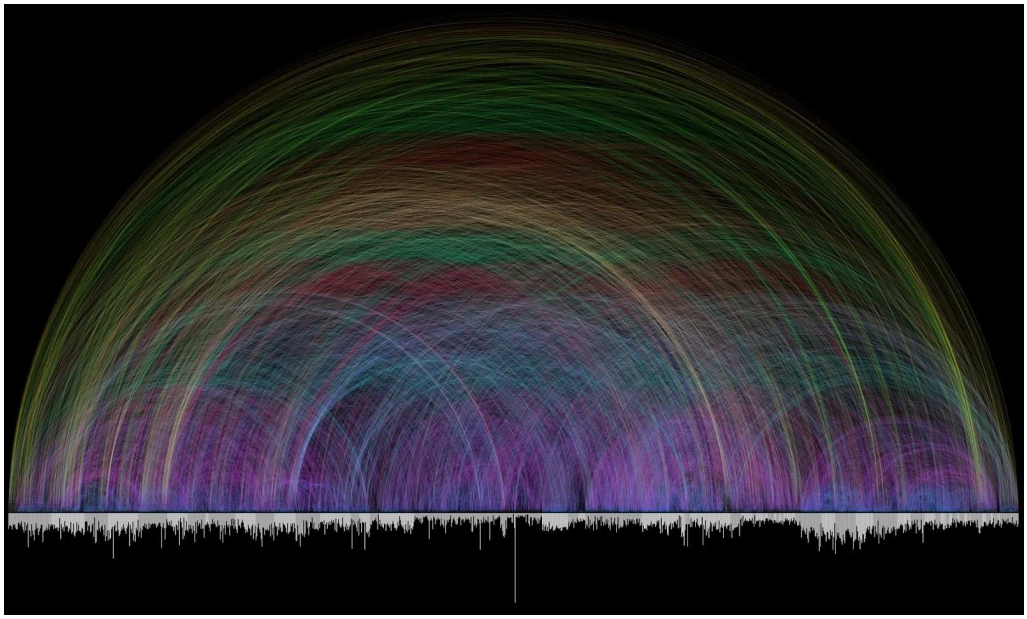

Fig.2.2.6 An arc diagram of cross references between chapters of the bible. [Carmi et al. 2007]

Arc diagrams originated from the work of Thomas Saaty [1964] who formally defined the minimum number of intersections of complete graphs drawn in 2 dimensions. All of the nodes of the graph are placed along a line in the Euclidian plane and connections are drawn as semicircles on either side of the two planes bounded by the line. Though it is more difficult to ascertain the structure of the graph, compared with a 2 dimensional layout “with a good ordering of nodes it is easy to identify cliques and bridges” [Heer et al. 2010].

2.2.3 Radial Convergence Diagrams

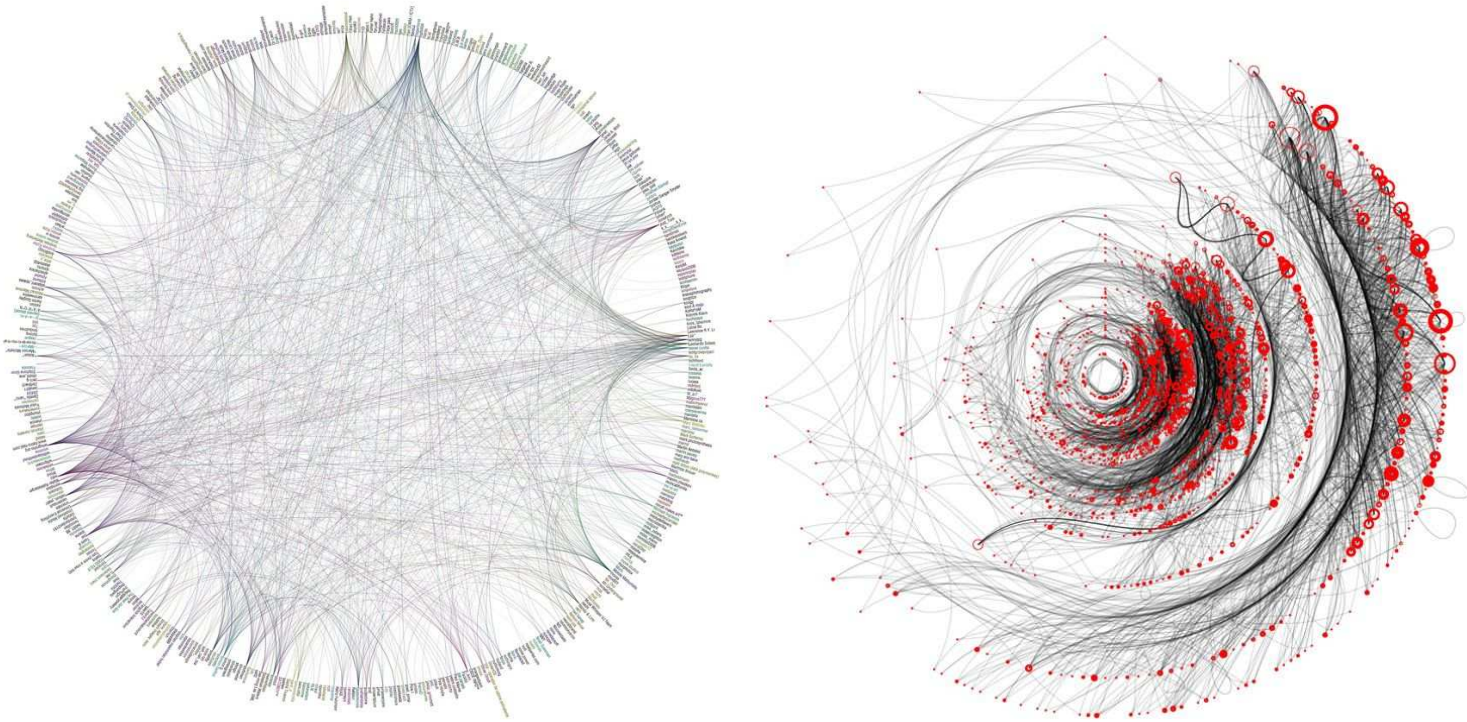

Fig.2.2.7 (Left) A visualisation of the connections between members of the Processing Flickr group. [Koberle 2007]
Fig.2.2.8 (Right) A segmented radial convergence visualisation of a poem. [Muller 2006]

Radial convergence diagrams, also referred to as chord diagrams maps nodes that are arranged radially around a circle with connections between the nodes mapped as arcs traversing within the circle. The advantage of a method like this is neutrality, by placing all of the nodes equal distances from each other this counters the tendency for viewers to assign importance to depicted in the center [Huang et al. 2007]. A variation on this arrangement is a segmented radial convergence diagram (Fig 2.2.8) which separates different sets of nodes into concentric circles, allowing subsets to be differentiated geometrically, as opposed to be differentiated with arbitrary labels or colours.

2.2.3 Flow Diagrams

The examples given thus far have all been visualisations of undirected graphical models, where the edges have no orientation. However, despite some notable exceptions [Courville 2015] most neural networks are directed graphical models. Therefore consideration for methods of visualising the orientation of connects must be taken, often in node-link diagrams arrowheads are commonly used to denote orientation [Di Battista et al. 1994]. But there are alternatives to this and flow diagrams are one such solution for visually displays interrelated information sequentially or chronologically. In the following subsections, two such types of flow diagrams that may be useful for visualising neural networks are described.

2.2.3.1 Sankey Diagrams

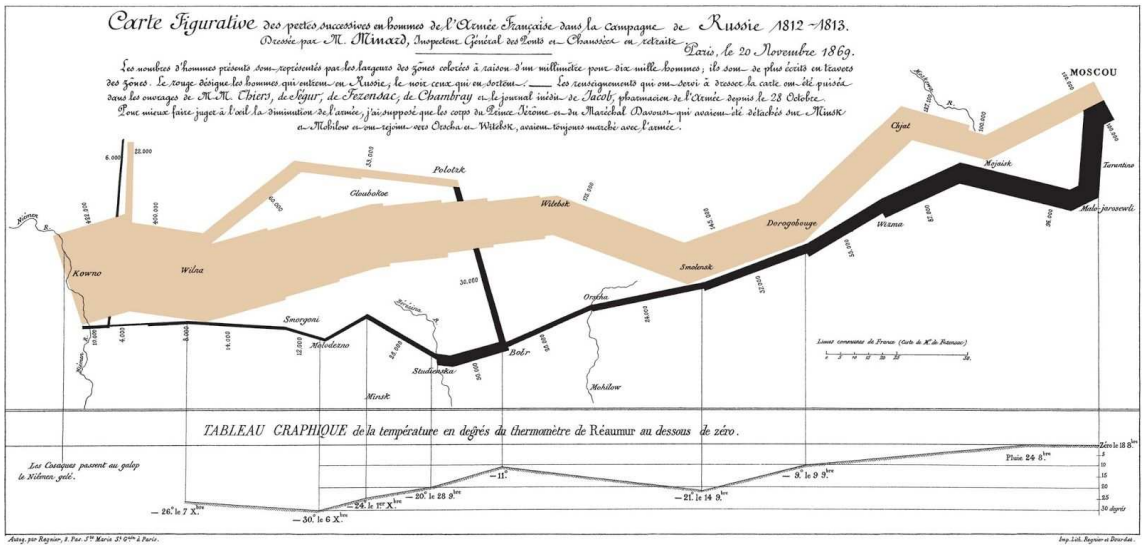

Fig.2.2.9 Charles Minard's 1868 diagram of Napoleon's disastrous Russian campaign of 1812

Sankey diagrams are type of flow diagram where the width of the lines represent proportionality of flow quantity and are typically used to visualise energy transfer or cost transfer between processes. Because the width of the lines is proportional to the total energy in the system, sankey diagrams give visual emphasis to the major flows or transfers of energy in a system.

2.2.3.2 Alluvial Diagrams

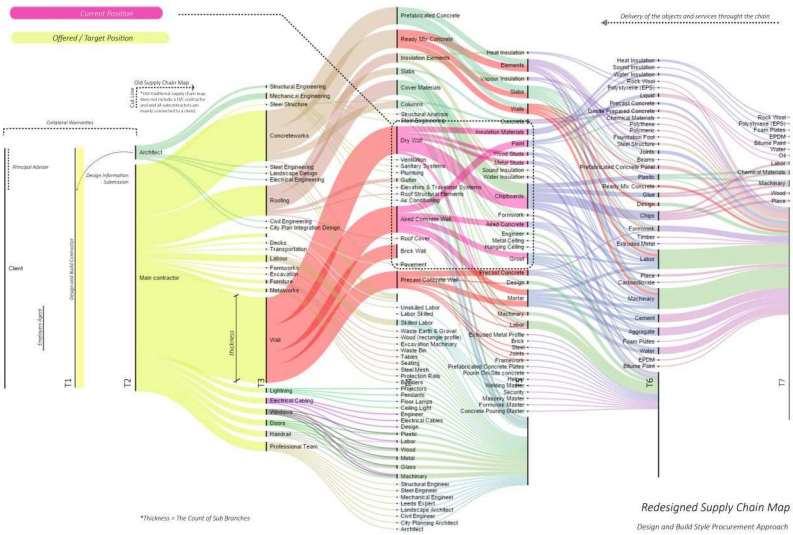

Fig.2.2.10 Alluvial Diagram of construction supply chain management. [Yildirim 2015]

Similarly to sankey diagrams, the width of lines in alluvial diagrams is used to represent proportionality, but alluvial diagrams can represent many overlapping or interconnect systems (as opposed to the energy flow of a single system). Alluvial diagrams were originally developed to visualize structural change in large complex

networks and can also be used to illustrate patterns of flow on a fixed network over time.

2.3 Visualisations of Biological Neural Networks

Creating a simulation of the human brain has been a long coveted milestone of development in artificial intelligence research [Kurzweil 1990, 2000, 2005 , 2012] if not one of dubious philosophical underpinning [Noë 2009, Dodig-Crnkovic and Müller 2011, Müller 2012]. It is a problem of massive scale and complexity (the human brain has 86,000,000,000 neurons and between 10^{14} and 10^{15} synapses). Despite the scale of this challenge, the European Union has funded €1.19 billion for the Human Brain Project in an effort to have a full simulation of a human brain by 2023 [Honigsbaum 2013]. Prior to the Human Brain Project, Henry Markram directed the Blue Brain project, where they performed accurate biologically simulations of neuronal behaviour of sections of rat brains [Markram 2006]. Out of this project many pioneering visualisations of biological neural networks were produced (See Fig.2.3.1).

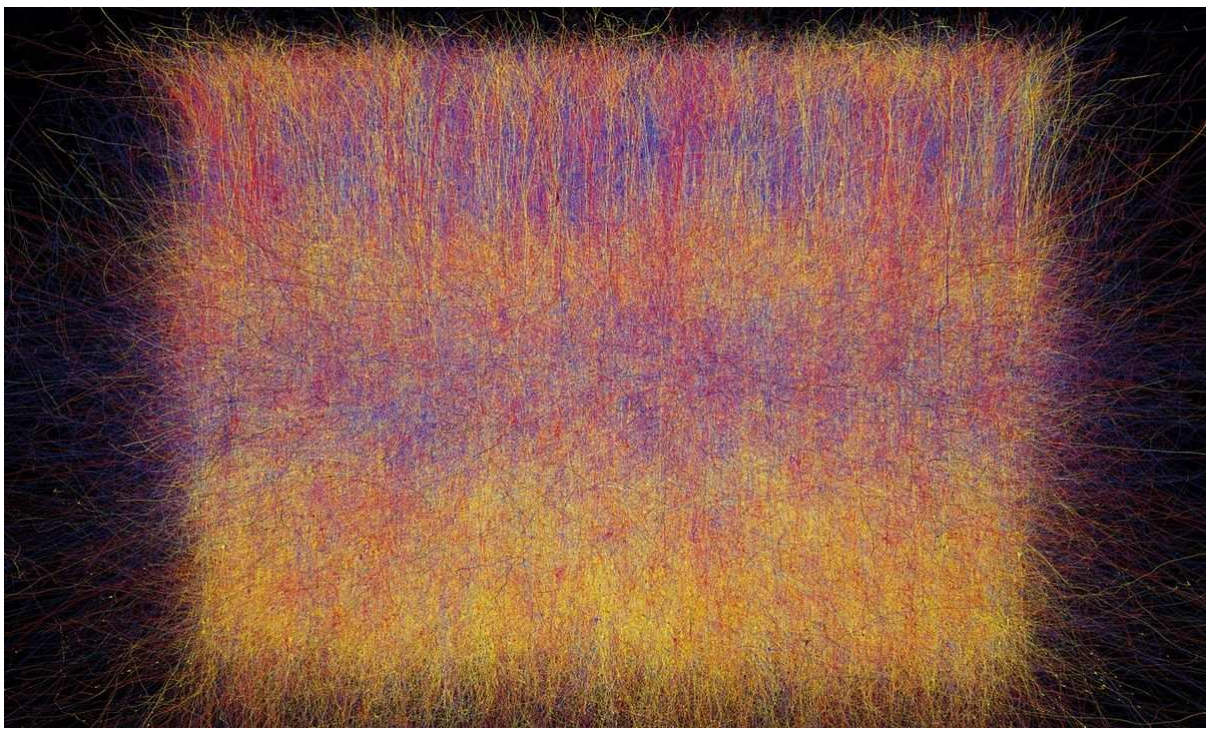

Fig.2.3.1 Visualisation from the blue brain project. [Markram 2006]

One of the problems with attempting to visualised massive biological neural networks is that there is too much information to adequately make sense of and visualise without much of the networks being completely obscured. One solution is to visualise small portions of the network that are key, perhaps, to understanding a specific mechanical process important from a neuroscientific perspective.

Alternatively, instead of visualising individual neurons on a microscopic scale, visualisations can be made of general flows of information on a macroscopic scale, such like the visualisations produced by Toga et al. [2012] in the Human Connectome Project. By tracking cerebral blood flow using various forms of brain imaging technologies, detailed constructions of the generalised topological structure of train brain can be mapped out (See Fig.2.3.2).

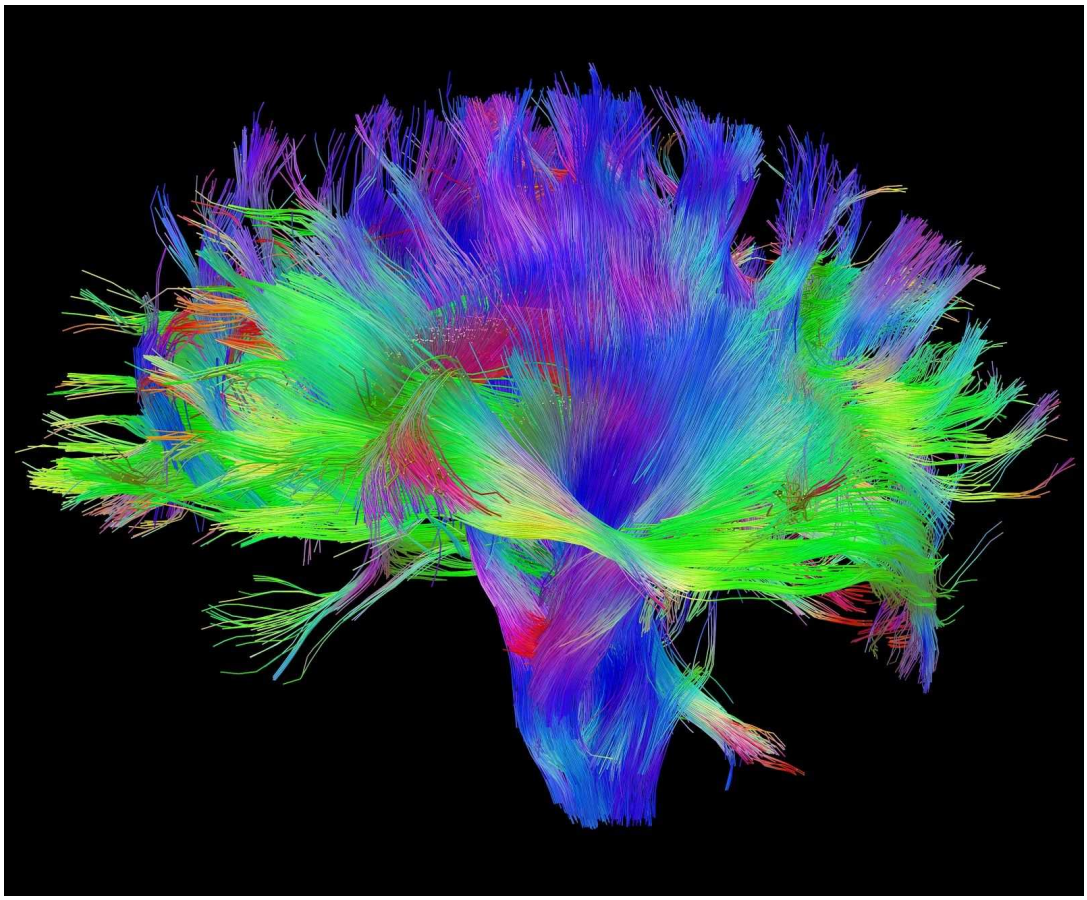

Fig.2.3.2 Visualisation from the human connectome project. [Toga et al. 2012]

2.4 Visualisations of Artificial Neural Networks

The purpose of this project is to produce both insightful and aesthetically pleasing visualisations of artificial neural networks. This section presents a survey of previous attempts to visualise artificial neural networks.

2.4.1 Visualising Unit Activations

One of the criticisms of artificial neural networks, specifically convolutional neural networks, is that despite their unparalleled capabilities at pattern recognition tasks, very little was known about the inner representations that the networks were developing, and whether they were in fact, accurate representations of objects akin to representations held by humans or whether some other form of convoluted and complex statistical reasoning was underlying these representation.

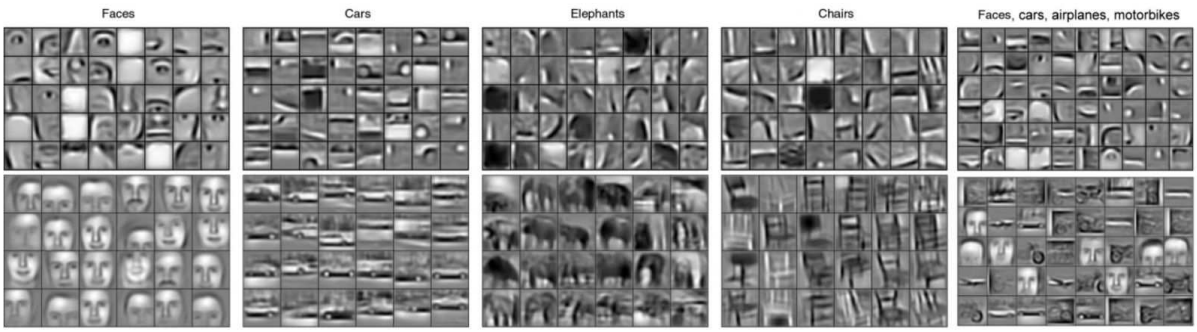

Fig.2.4.1 Learned representations in convolutional deep belief networks visualized as a weighted linear combination of lower layer bases. [Lee et al. 2010]

Honglak Lee et al. [2010] were the first to visualise these learned representations in deep layers of these networks (See Fig.2.4.1), by combining filters from low layer bases as weighted linear sums. Zeiler and Fergus [2014] took this method further by using deconvolutions and filtering the maximal activations, one can find the approximate purpose of each convolution filter in the network. They were also able to pair these to image segments presented to the network that maximally activated particularly convolution filters (See Fig.2.4.2).

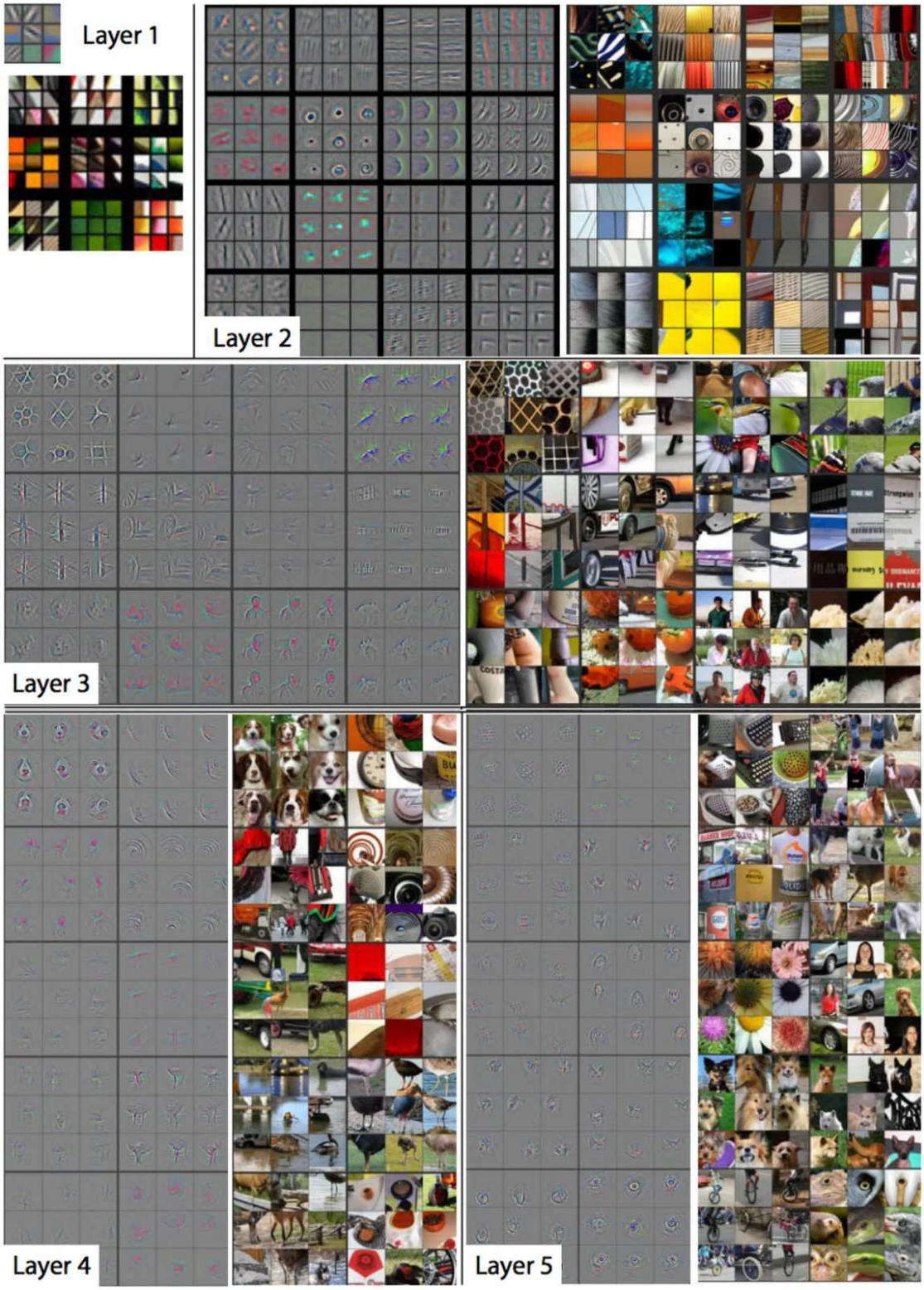

Fig.2.4.2 Visualisations of different 'neurons' at progressively higher levels of representation, alongside the sections of images that each 'neuron' is most responsive too. [Zeiler et al. 2014]

2.4.2 Topological Visualisations

Visualisations of filter activations have been done very effectively by Zeiler and Fergus [2014] and gives great insight into inner representations of the various convolutional filters in these networks. What has been done less effectively is visualising the topological structures that also contribute to the ability of these networks to perform sophisticated pattern recognition. This subsection briefly overviews the challenges, and a survey of some of the examples of solutions to these challenges.

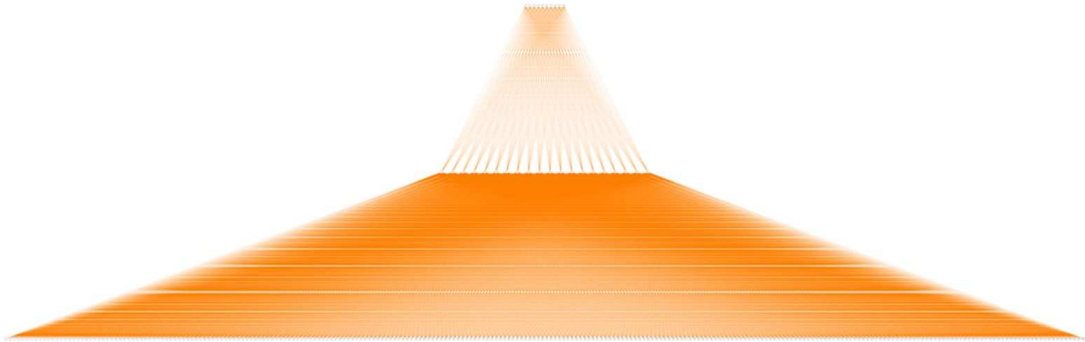

Fig.2.4.3 A fully-connected network for classifying digits from computer fonts. [Harley 2014]

One of the obvious problems to visualising modern artificial neural networks is the sheer scale of these networks. There may be thousands, or even millions of connections between nodes, and visualising all of them may be challenging due to computational constraints, or the constraints of the human visual system to process such a vast amount of information. In addition to this, naively drawing all connections would most likely obscure any possibility of people perceiving the 'important' connections (especially where there are many layers that are fully connected).

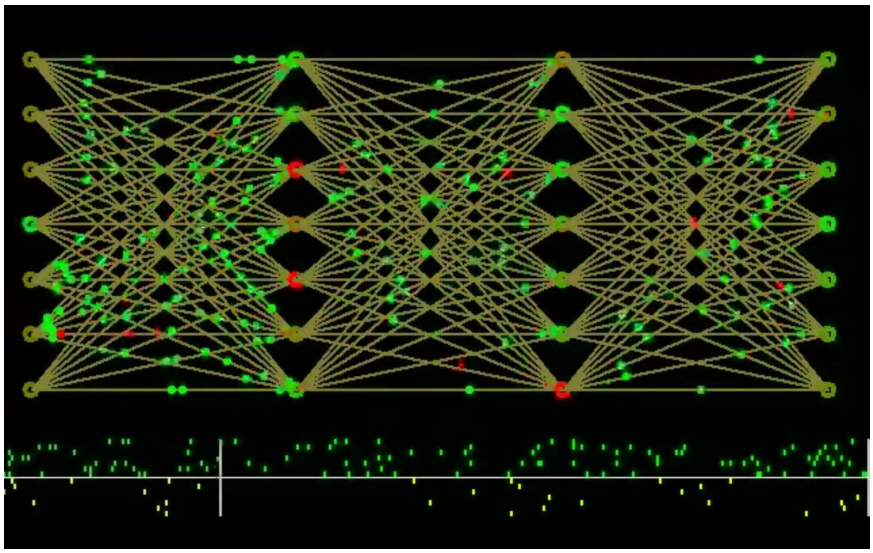

Fig.2.4.4 Spiking Neural Network v.1.1. [Roberto 2010]

One obvious solution is to make visualisations that are small enough so that all of the nodes and connections can be seen clearly and that there are not too many edges that are overlapping and obscuring each other (such as Fig 2.4.4). The problem with this approach is that in all likelihood the network model will be so small that it it is not processing data of any significance, or that the model is not big enough to build a statistical model of significant complexity that it would be performing any act of reasoning that would be considered interesting to the viewer. In addition, there is a tendency for these models to be toy networks, that are technically fully functioning neural networks, but are not actually trained on any real datasets and are not performing any kind of meaningful pattern recognition at all (see Fig.2.4.5).

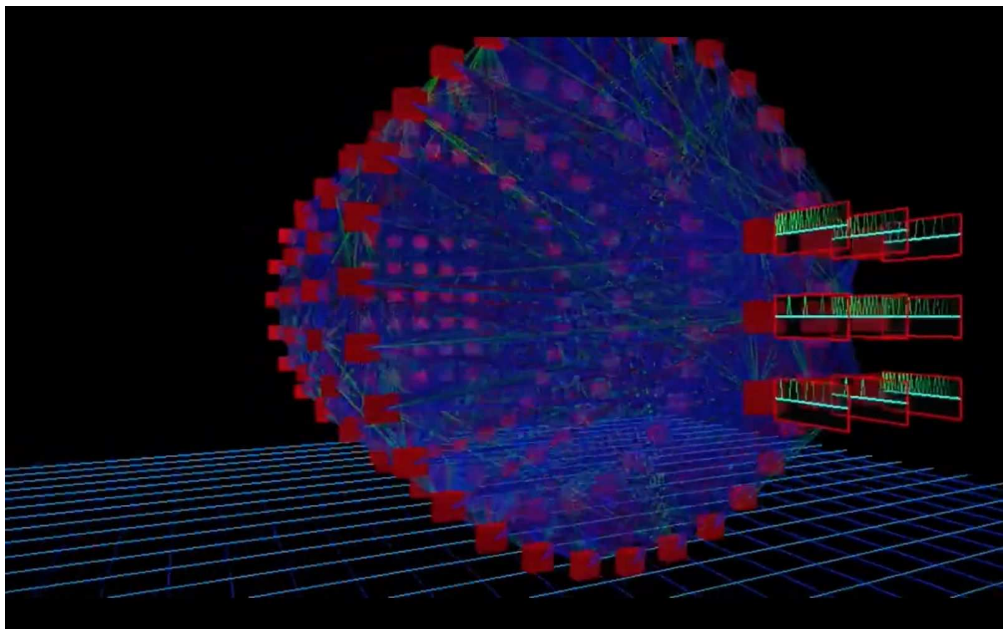

Fig.2.4.5 Neural network simulator. [Ortega 2007]

Contrary to the approach of visualising small 'toy' neural networks, the aim of this project is to visualise the types of neural networks that perform the type of meaningful and complex pattern recognition tasks (such as CNN's and LSTM's) that have produced such a flurry of excitement for the potential of these techniques. To visualise the inner workings of these much larger and more complex networks, a more selective and nuanced approach must be taken.

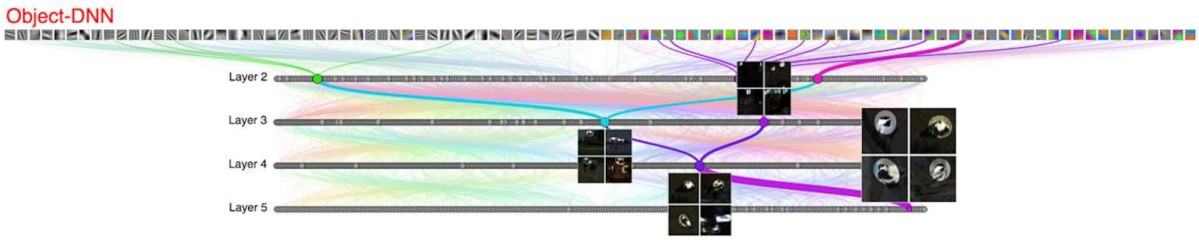

Fig.2.4.6 Drawnet: Deep Neural Networks predict Hierarchical Spatio-temporal Cortical Dynamics of Human Visual Object Recognition. [Cichy et al. 2016]

Cichy et al. [2016] have done an excellent job of representing a meaningful depiction of the hierarchical topology contributing the image recognition in convolutional neural networks (see Fig.2.4.6). By combining visualizations of the filter activations and highlighted image segments of particularly responsive image regions for convolutional filters (a technique popularised by Zeiler and Fergus [2014], see section 2.4.1) with alluvial diagrams that by using width (see section 2.2.4.2) depict the proportional data flow that contribute to each filter activation. An equally effective alternative to this approach is Adam Harley's [2015] web based 3D interactive node-link visualisation of a convolutional neural network (see Fig.2.4.7). Because it is interactive only the connections contributing to the activation of the node that the user hovers over are visualised, avoiding the problem of rendering too many connections and thus obscuring the viewer's ability to perceive important topological structures. Harley's visualisation also allows you to draw your own input for the network (the network being a convnet trained to recognise numbers from the MNIST digit database), engaging the viewer interactively, and allowing the viewer to test the ability of the network to deal with inputs that are not from handwritten digits.

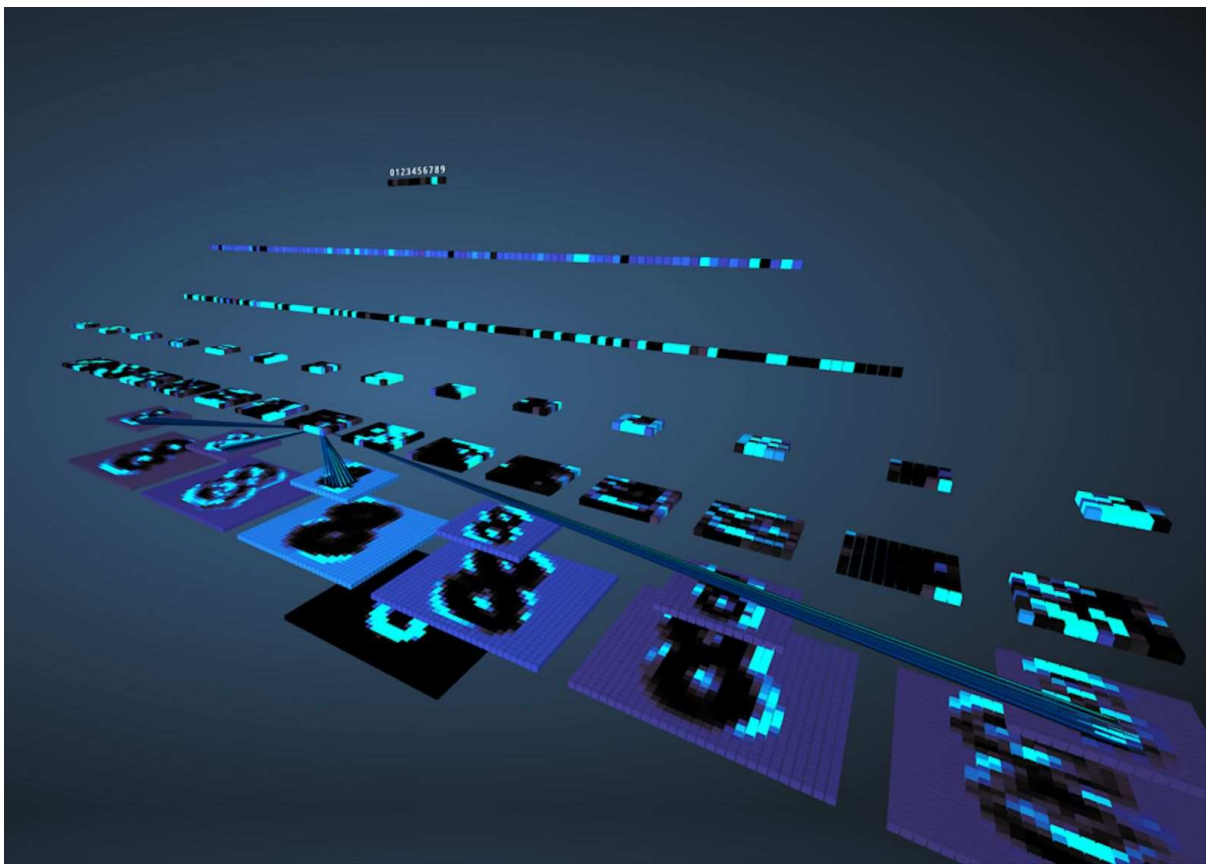

Fig.2.4.7 An interactive node-link visualisation of a convolutional neural network. [Harley 2015]

3 Research Questions

3.1 Aim

The aim of this project is to develop a method for visualising the topological structures that emerge in artificial neural networks when they are exposed to data structures that the networks have *learnt* to recognise, for both aesthetic purposes and to gain insight into the inner workings of artificial neural networks. Appropriate techniques from the field of data visualisations will be used and adapted for the purposes of this project.

A secondary goal for this project is to produce animated visualisations that bring to light the progressive convergence of these topological structures as they take form during the training process. Exposure and comparison could be given to different learning algorithms (such as stochastic gradient descent and RMSProp) regularisation techniques (such as dropout). In addition to this animations of neural networks that inherently exhibit dynamic temporal behaviour, such as recurrent neural networks, could offer great insight into the mechanics of such networks, since current there have been no such visualisations of these networks.

3.2 Motivation

Artificial neural networks are a fascinating field of research, but they can be notoriously difficult to comprehend as they are described as these structures of many layers of interconnected volumes of artificial ‘neurons’. In reality the structures of these networks are very regular, and straightforward enough to understand, just that they are defined in abstract terms by complex mathematical notations, thus only accessible to persons of a certain expertise in dealing with and understanding this literature. Visualisations take advantage of the human visual system's vast capacity for processing and comprehending complex patterns, and help give people insight into complex abstract structures that they would otherwise have difficulty processing. In addition to this I believe there is a lot of artistic potential in this project, deep learning has become very trendy recently and their is palpable excitement for artistic projects that incorporate neural networks [Clark 2015]. There is also a huge number of possibilities as there are many architectures and configurations of neural networks to choose from as well as many different aspects of these to emphasise with the visualisation.

3.3 Problems to Overcome

Modern day neural network architectures are often massive structures with many thousands of nodes with sometimes millions of interconnections. Quite often these layers of nodes are fully connected between each other, creating huge amounts of overlapping edges in visualisations (see Fig 2.4.3). If one was to naively plot of the edges in a neural network such as the one described, any ability to perceive which connections were important would quickly be lost through important edges in the graph being obscured by the vast volumes of redundant edges. The challenge for this project is to find a method of visualising or only drawing attention to the important edges in a network, that contribute to the topological structures of activation involved in the process of pattern recognition. I believe appropriating some of the characteristics of sankey and alluvial diagrams will be useful for this problem, as has already been successfully explored by Cichy et al. [2016] (see Fig 2.4.6), but there is huge unexplored potential in taking this approach and applying it to 3D visualisations. There are also alternatives to representing proportionality of magnitude as width, both colour and transparency could be used, as well as thresholding so as only to display connections that are carrying values above a certain designated threshold.

From a technical standpoint, one of the biggest difficulties and perhaps reasons why this kind of visualisations has not been attempted much, is that modern day machine learning frameworks are optimized for efficiency and do not give access to much of the computations being performed under the hood. Often values of the weights interconnecting layers, and the layers of units themselves are defined as high dimensional tensors in high level programming language such as python, and the execution of these computations is performed in a low level language such as C or FORTRAN [Abadi et al. 2015]. This makes it very difficult to access the values of the weights in these networks, and even harder to ascertain the values flowing down these connections when the network is exposed to a data sample. The only machine learning framework that does give easy access to all of these values in ConvNetJS [Karpathy 2014]. The problem with ConvNetJS is that it is based in javascript so it may not be suitable for very large scale and high resolution visualisations (that would possible need to be pre rendered) , and that it is not a general purpose machine learning library. ConvNetJS is only suitable for small scale convolutional neural networks, and novel examples of a couple of other network types. It is not capable of modelling recurrent neural networks, which is one of the types of networks that I am aiming to visualise for this project. The alternative solution is to build these neural networks completely from scratch, in a programming environment suitable for visualisation (like C++ or javascript), but this inevitably incurs many other challenges and would only be done if absolutely necessary.

4 Method

4.1 Building the Neural Network

After evaluating several options ConvNetJS was chosen for the architecture and training of the neural network. ConvNetJS was the most suitable for several reasons: The source code is open, accessible and easy both to read and edit. Pre-trained networks can be saved to JSON, and new networks instantiated from the JSON snapshots. Javascript is an ideal environment because of it is speed, ubiquity, and the ease of which the neural network processing can integrated with WebGL and interactive elements.

For the purposes of this project the network that will be visualised is the standard and ubiquitous convolutional neural network designed for classifying MNIST digits. These networks trained to classify MNIST have a long history dating back to Yann LeCunn's [1998] LeNet 5, and having become something akin to the "Hello World" of machine learning. Because of the simplicity of the dataset (small resolution and one colour channel) it is the ideal starting point for experimenting with this kind of visualisation.

The network architecture is as follows:

1. Input Layer [Width - 24, Height - 24, Depth - 1]
2. Convolutional Layer [Width - 24, Height - 24, Depth - 8]
- [Filter Size - 5x5]
3. Rectified Linear Unit Layer [Width - 24, Height - 24, Depth - 8]
4. Max Pooling Layer [Width - 12, Height - 12, Depth - 8]
- [Pooling Size - 2x2]
5. Convolutional Layer [Width - 12, Height - 12, Depth - 16]
- [Filter Size - 5x5]
6. Rectified Linear Unit Layer [Width - 12, Height - 12, Depth - 16]
7. Max Pooling Layer [Width - 4, Height - 4, Depth - 12]
- [Pooling Size - 2x2]
8. Fully Connected Layer [Width - 1, Height - 1, Depth - 10]
9. Softmax Classifier Layer [Width - 1, Height - 1, Depth - 10]

4.2 Storing All of the Values From a Forward Pass

To differentiate this project with other neural network visualisations, visualising the values being passed down the connections given a learned data type is the core aim of this project, so it is necessary to store every value being passed down every connection in a forward pass of the network. This information is not normally stored, as it is unnecessary and inefficient. In ConvNetJS only the weight matrices, and output node values are stored in arrays. Therefore a data structure was created to store all of these values, and edit the source code of ConvNetJS in order to store these values in a forward pass of the network.

In ConvNetJS weight matrices and node values are stored with the `Vol` data structure. It is essentially just a fast typed javascript linear array, with functions to store and retrieve elements in the array with the indices `x`, `y` and `d`. Allowing it to be treated as if it were a 3rd-order tensor with the depth of the X,Y and Z dimensions being defined with the variables `sx`, `sy` and `depth`. Retrieving an element from the linear array from the given indices is calculated using this formula :

$$index = (sx * y) + x) * depth + d$$

Since both the node values and the weight matrices are stored in these 3rd-order tensor volumes, the `Vol` data structure was not going to be high enough of a dimensionality to index and store the value of every weight connection going into every node. Therefore the volume data structure paradigm is extended to create a volume of volumes, named `VolofVol`. This data structure is indexed in exactly the same way as Vol and is infact very similar, the only difference being that instead of a floating point value being stored in each element of the array, a Vol object is stored instead. And of course there is more values used to index and retrieve values `x`, `y`, `d` to index the `Vol` object and `vx`, `vy`, `vd` to index the value within the `Vol` object.

To then store each value a `VolofVol` object is added to the object definition of each layer. Then in the forward pass function for each layer the value being carried down each connection - as defined by the connection weight multiplied by the output value of the node from the previous layer - is stored in the the `VolofVol` object like so:

```
activation = f.w[((f.sx * fy)+fx)*f.depth+fd] * V.w[((V_sx * oy)+ox)*V.depth+fd];
this.VolofVolofActivations.setVal(ax,ay,d,fx,fy,fd,activation);
a += activation;
```

Whereas in the original ConvNetJS code the loop through every input of every node in a layer would have just had the one line of code:

```
a += f.w[((f.sx * fy)+fx)*f.depth+fd] * V.w[((V_sx * oy)+ox)*V.depth+fd];
```

With `a` representing the given equation once a full loop through all of the inputs is completed:

$$a_L^i(x) = \sum w^i n_{L-1}^i(x)$$

Where node *a* at index *i*, layer *L* is the nonlinear weighted sum of the input nodes n in the previous layer given an input *x*.

4.3 Rendering The Visualisation

The library Three.js was chosen for rendering the visualisation. It is a library simplifying WebGL, and is built specifically for creating simple 3-dimensional objects and rendering small 3-dimensional scenes.

4.3.1 Rendering The Nodes

In ConvNetJS activation functions are represented as being their own layers, separated from the previous layer (which is most like a weighted sum, as in the network being visualised is either a convolutional layer or fully connected layer). Whereas in the research literature and other machine learning frameworks the activation function and weighted sum of the inputs are represented as one layer of nodes. Therefore Softmax and ReLu (rectified linear unit layer) are not going to be rendered as separate layers, but instead use their values to colour the nodes of the fully connected and convolutional layers.

The first step in initialising the positions of all of the nodes is to loop through every layer in the network, and create a new array called `layerLookup` that pushes the value of the index layer to that array if it is a layer type that is going to be rendered (every layer type except ReLu and Softmax). There is a nested set of for loop in the function `initPosArray()` that loops through the length of the `layerLookup` array and grabs the layer at the index given in the `layerLookup` array, unless it is a convolutional or fully connected layer, then it grabs the index given from the `layerLookup` array plus one to retrieve the following Softmax or ReLu layer (that always has the same dimensions).

The next step is to - nested within each loop - loop through the depth (Z dimension) of the network layer followed by looping through the X dimension (image

width) and the Y dimension (image height). The position of each node in relation to the origin is then calculated by the `getPos()` function that equates to:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ((d * width * -1.5) - (depth * width * -1.5)/2) + (x * -1 - (width * -1)/2) \\ y * -1 - (height * -1)/2 \\ d * 100 - (depth * 100)/2 \end{bmatrix}$$

Unless the layer is a MaxPool layer or a Fully Connected layer in which case the function is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ((d * width * -3) - (depth * width * -3)/2) + (x * -1 - (width * -1)/2) \\ y * -1 - (height * -1)/2 \\ d * 100 - (depth * 100)/2 \end{bmatrix}$$

So as to space align the center of the MaxPool layers with the center of the preceding Convolutional layers, and the spread out the nodes of the fully connected layers to make the visualisation easier to read. Later it was decided that although the input layer has a depth of one (in the case of MNIST digits, otherwise usually three if it is a colour image), it would be easier to read visually if their was one input for each convolutional filter in the first Convolutional layer. Therefor for the equation above, the d and depth values are substituted in from the following Convolutional layer.

Each node is simply represented as a cube with a width, height and depth of 1. The colour is simply extracted from the `Vol` in each layer called `A` that stores the node output after each forward pass. In ConvNetJS the image input values are in the range -0.5 to 0.5, therefore before assigning the value from `A` to the colour value of each cube, 0.5 is added to the value extracted from the node to set the colour. The colours could be better normalised for each layer (as in the upper layers the range the range can exceed -0.5 and 0.5), but that would require a forward pass through every node, and for the sake of efficiency as it is in javascript this is not performed. Fig 4.3.1 shows all of the nodes drawn in 3D space with the colours assigned to them from the forward pass.

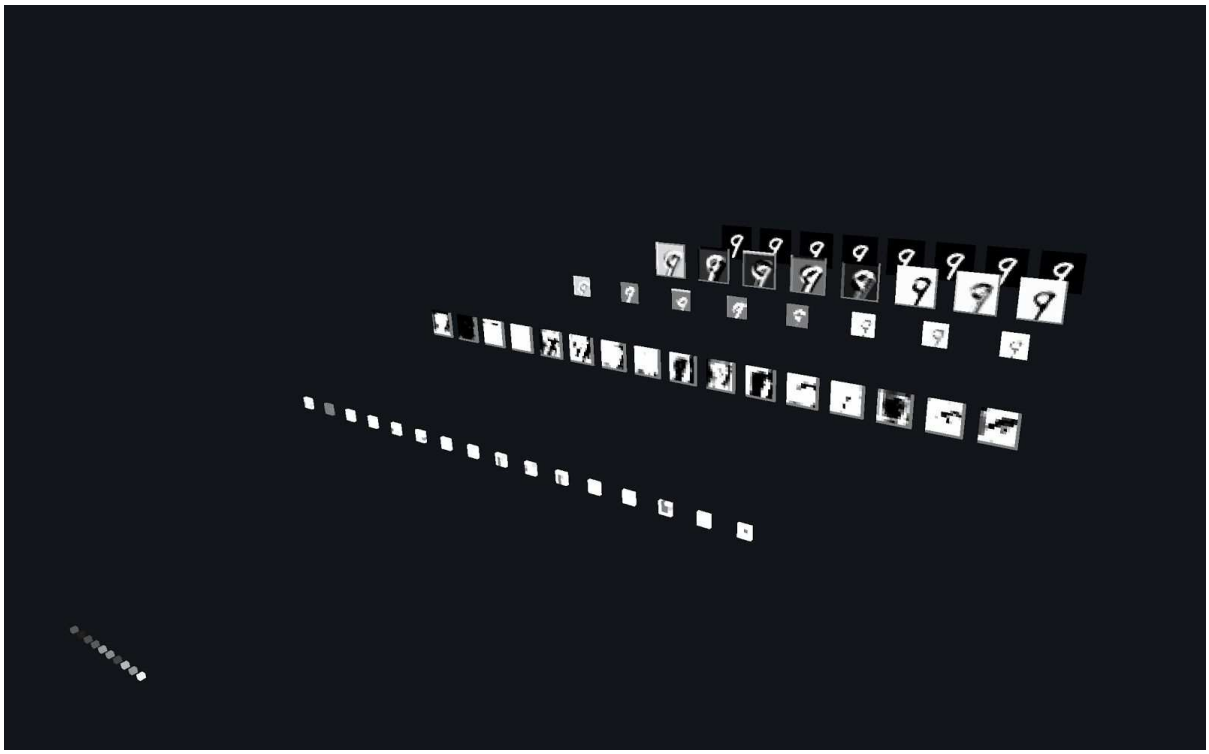

Fig.4.3.1 A screenshot of all of the nodes being rendered, with the colours assigned from the outputs after a forward pass.

4.3.2 Determining Which Connections To Render

The crucial difference between this visualisation and previous attempts to visualise neural networks is not just that the value passed down each connection from a forward pass is being visualised, but that connections are selectively chosen to be rendered as being important in the context of digit classification. The simplest way to determine whether an input connection is carrying import information is to see if it is above a certain positive threshold, if it is, then if it is obviously an important feature that could be contributing to a high positive node output. However not every connection that is above a certain positive threshold is going to necessarily contribute to a correct classification. Therefore we need to find the positive connections that do contribute to the correct classification.

4.3.2.1 Recursive Depth-First Tree Search

To find the connections (and thus the nodes) that contribute to a correct classification we need to search from the highest value node in the top layer of the network (Fully Connected followed by Softmax), search the value of each input connection after the forward pass to decide which connections to render. After that we need to repeat the same process for the next layer, rendering the connections above a certain threshold contributing to the nodes that have already had connections drawn to them from the previous layer. This process needs to be

repeated for each layer until the connections to the input layer are drawn. The algorithm just described is breadth-first tree search, whereas the algorithm implemented in the project is depth-first tree search. Depth-first tree search was chosen as it can be more elegantly implemented with recursive functions, without the need for separate arrays indexing which nodes need to be searched in the next layer, and with the ability to draw in the connection and then call the function to search the node in the next layer within the same function call. The small downside to this approach is that sometimes the same connections get drawn multiple times, but given the size of the network and the amount of connections drawn, it was not deemed necessary to rectify this inefficiency.

To implement the function `treeSearch()` it takes a custom data structure (javascript object) `index` as its input parameter. The object has 4 values assigned to it, layer index (`l`), depth index (`d`) , width index (`x`), and height index (`y`). The function `treeSearch()` is infact a simple switch statement, for layer index `l` it looks up the layer type, if the layer type is Softmax or ReLu it simply calls the function `treeSearch()` again but the with layer index `l` in the `index` object with a value of `l-1`. This is because, as stated earlier, layers that are simply activation functions are not being rendered in this visualisation. In the case that the layer index `l` is `0` and represent the input layer no further functions are called as this is the final layer in the network.

In the case that the layer is Fully Connected, Convolutional or a MaxPool layer the respective functions `treeSearchFC()`, `treeSearchConv()` or `treeSearchPool()` are called given the same index parameter. There are three different functions because in ConvNetJS the index for which node in the previous layer is used for the node output in the MaxPool layer is already stored in the arrays `switchx[]` and `switchy[]` in the original ConvNetJS source code (*along with the comment "`Store pointers to where the max came from. This will speed up backprop and can help make nice visualizations in future.`"*). Therefore these values did not need to store these Values in a `VolofVol` object. The `treeSearchPool()` function simply finds the the `x` and `y` indices of the input node from the `switchx[]` and `switchy[]` arrays, creates the a new index object where the layer index is one less, the depth is the same, and the x and y indices are those just retrieved from the `switchx[]` and `switchy[]` arrays. Then the function `drawLine()` is called (see 4.3.3) given the `newIndex` parameter and the `index` parameter that was passed into the `treeSearchPool()` function when it was called. After calling the drawLine() function (which draws a line between the two nodes) the function `treeSearch()` is then called and given the index for the next node, thus the process is repeated recursively.

The reason there are separate tree search functions for Fully Connected layers and Convolutional layers is because the dimension of the connections that connect into the nodes in these layers is different. In Fully Connected layers the dimension of the input volume `Vol` stored in the object `VolofVol` (one `Vol` for each node in the layer) has the same dimensions as the preceding layer (as it is connected to every node in the previous layer). In Convolutional layers the x and y dimensions of the connections that connect into the nodes is the filter size of the layer (in the case of this network both Convolutional layers have a filter size of 5x5) and the depth of the previous layer. Therefore substantially different methods for searching the connection values in the volume `Vol` retrieved from the `VolofVol` object, and then retrieving the corresponding node index from the previous layer, required that different functions were made for what is in essence the same process. Great care was taken to ensure the order of the nested for loops for searching the volume `Vol` (which is different for Fully Connected and Convolutional layers) was done in the same order that is used in the forward pass functions in ConvNetJS, so as not to accidently retrieve the wrong index for the node in the next layer.

Despite the differences, at the core of both functions `treeSearchFC()` and `treeSearchConv()` the same process is carried out, if the connection value is higher than an arbitrary threshold (in the case of this visualisation it is 0.2) the index for the node in the preceding layer corresponding to the connection is calculated; for the Fully Connected layer it is simply the index of the connection (with the layer index `l` subtracted by one), for the Convolutional layer it is node indices `x` and `y` subtracted by the padding width and added by the filter indices `x` and `y`, and the filter depth index `d` (with the layer index `l` subtracted by one). The function `drawLine()` is then called with the node `index` and the index of the corresponding node `newIndex`. Then function `treeSearch()` is called given the new node index `newIndex`. In the case that we are drawing multiple inputs to correspond to the number of filters in the first convolutional layer, the depth index `d` of the `newIndex` is the same as the node index.

4.3.3 Rendering The Connections

As described in the previous section, in the tree search functions when a connection is above a certain threshold (or pointer retrieved from the MaxPool layer) the function `drawLine()` is called given the respected `index` objects that define the nodes in the network. Originally all of the position vectors were initialised and stored in a 4-dimensional nested array, that could then be later retrieved indices `l`,`d`,`x` and `y` for the node. This however caused too many fatal exceptions when an index that exceeded the dimension of one of the arrays was accidentally called. Instead it was easier simply to recalculate the position using the `getPos()` function that is described in section 4.3.1. Initially `drawLine()` was implemented to draw a straight line between the two nodes as can be seen in Fig 4.3.2.

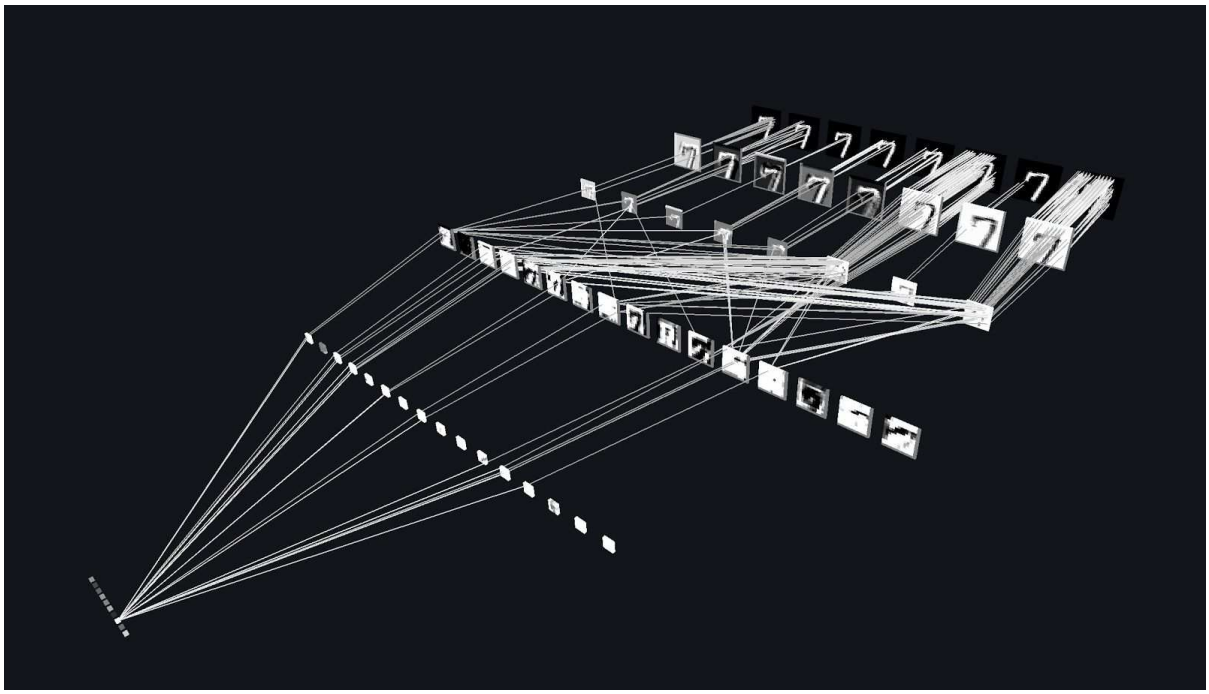

Fig.4.3.2 A screenshot of all the connections being rendered as straight lines.

Representing the connections as straight lines is what is most commonly done in previous neural network visualisation, and although it is does effectively portray the connections between the nodes (and is probably the most efficient way of visualising a large network), it can be quite difficult to make out the connections between nodes when there is a lot of overlapping connections. In addition to this it is not as aesthetically pleasing as some of the network visualisations that have curved lines between the connections, in particular the curved lines that are used in alluvial diagrams (see Fig 2.2.10) that are used to represent the flow of energy into an out of the different classifications in each layer. The reason they are so effective for representing this flow of energy is because the curved connections enter and exit at an angle orthogonal to the layer. This makes it easy to visually trace the path of the energy flow between layers.

Implementing the rendering of curved lines that enter and exit orthogonally to the each layer turned out to be quite simple in Three.js. The easiest way to define this kind of line is using a cubic bezier curve, with the origin and end points being the origin of the node positions defined by the `getPos()` function that is described in section 4.3.1, and the control points respectively being in the same positions except for in the z dimension (the axis along which the layers are spread out) in which the z position for both control points being along the midway plane between the two layers. This is done calling the function `THREE.CubicBezierCurve3()` with the first and last parameter being the origin and end position of the curve, and the second and third parameters being the control points for the origin and end position respectively. The vertices for the series of points along this curve is called using the

`curve.getPoints()` function, in the case with the parameter being 50. This is then converted to a Mesh object and added to the scene. Fig 4.3.3 shows the rendering with cubic bezier curves.

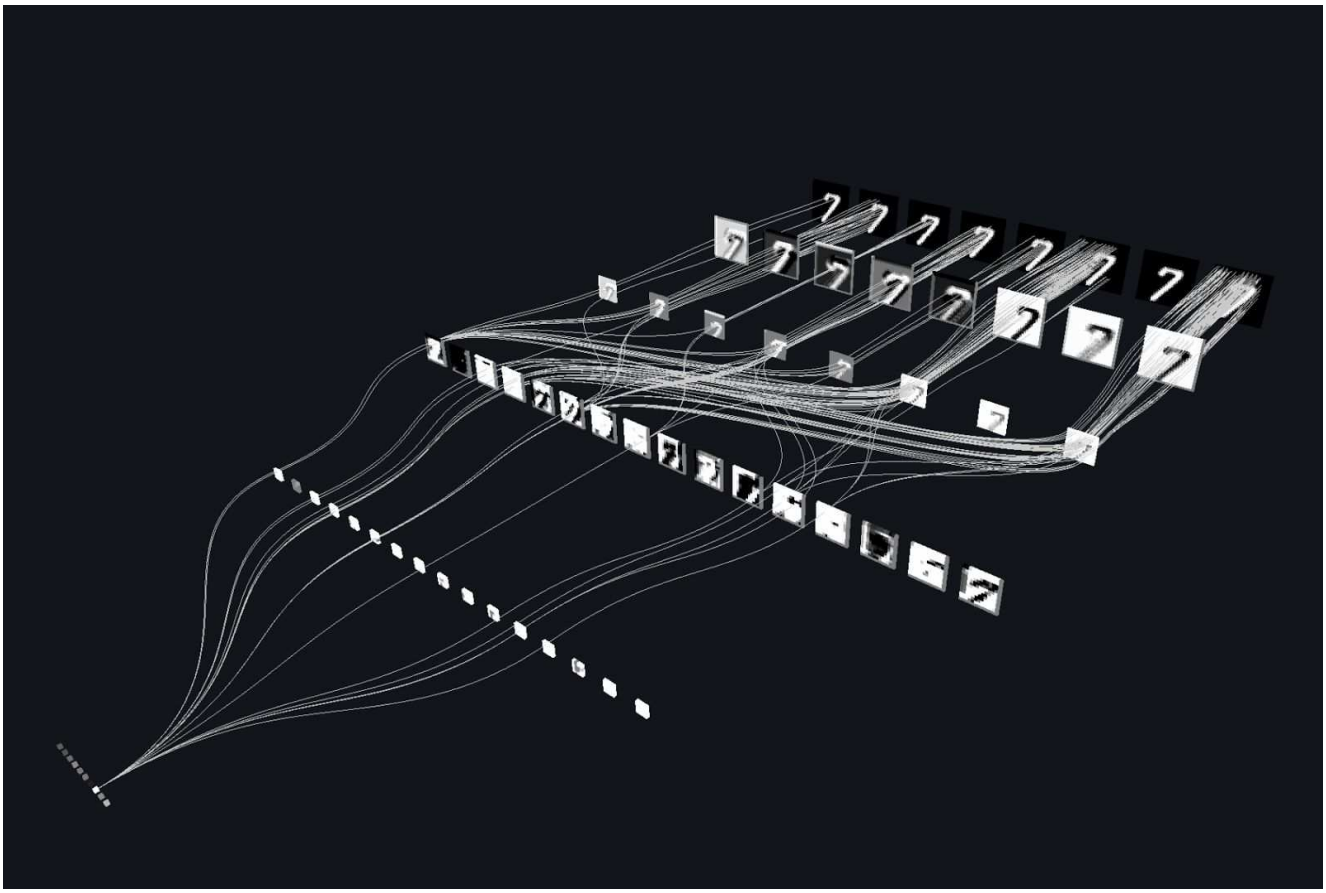

Fig.4.3.3 A screenshot of all the connections being rendered as cubic bezier curves.

5 Results

This chapter simply shows the results from the visualisation from different angles and then showing classification of every digit respectively. To access the visualisation online, it is available at the url:

http://terencebroad.com/convnetvis/vis.html

5.1 Different Viewpoints

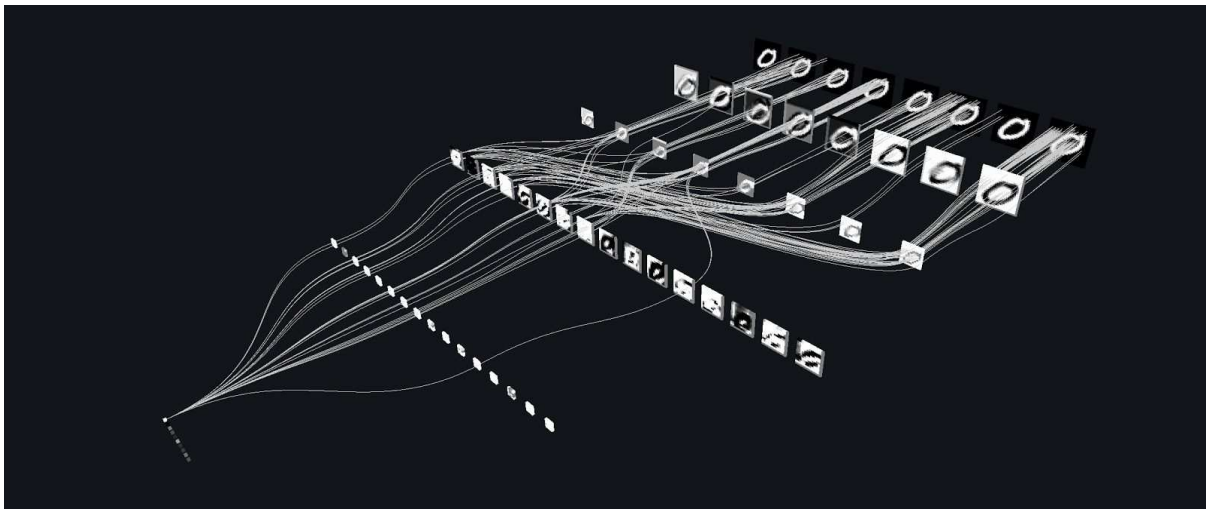

Fig 5.1.1 Off center view from above.

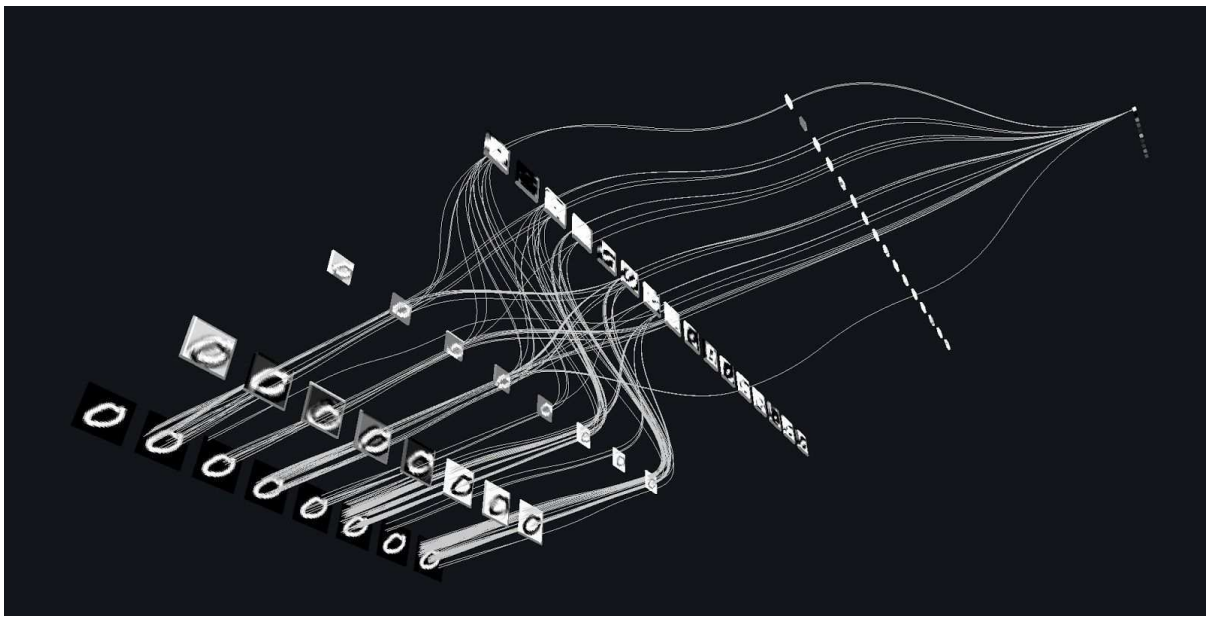

Fig 5.1.2 Off center view from below.

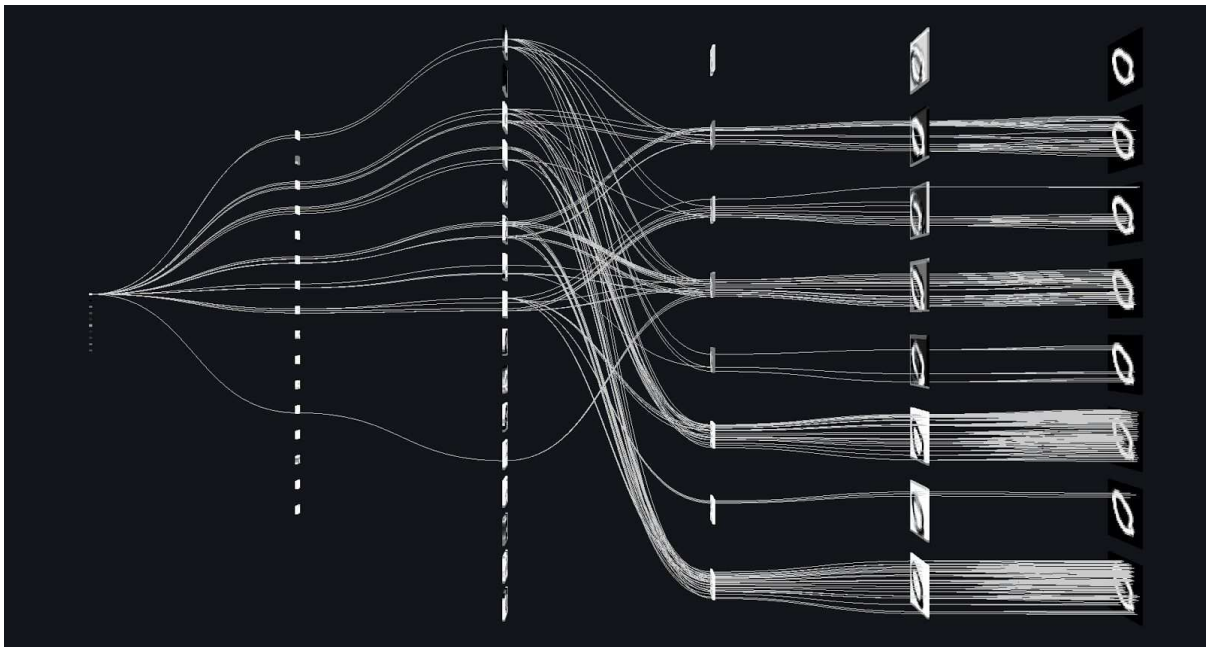

Fig 5.1.3 View from above.

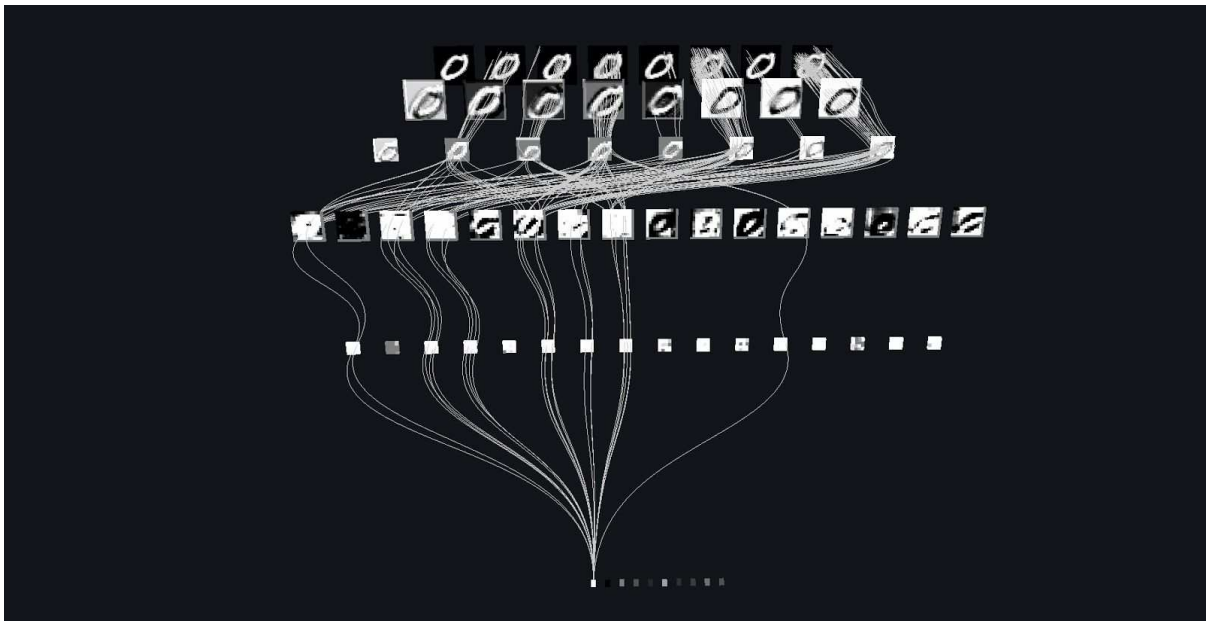

Fig 5.1.4 Central perspective viewed towards the network.

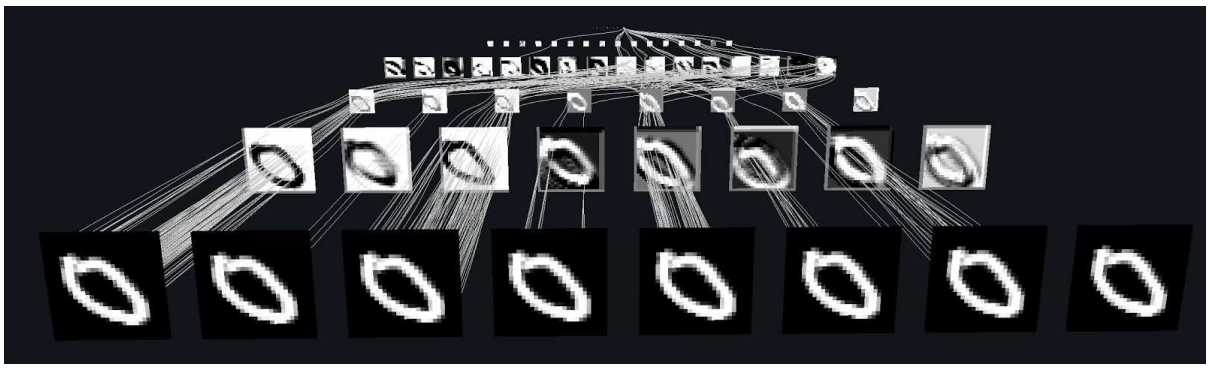

Fig 5.1.5 Central perspective view from behind the network.

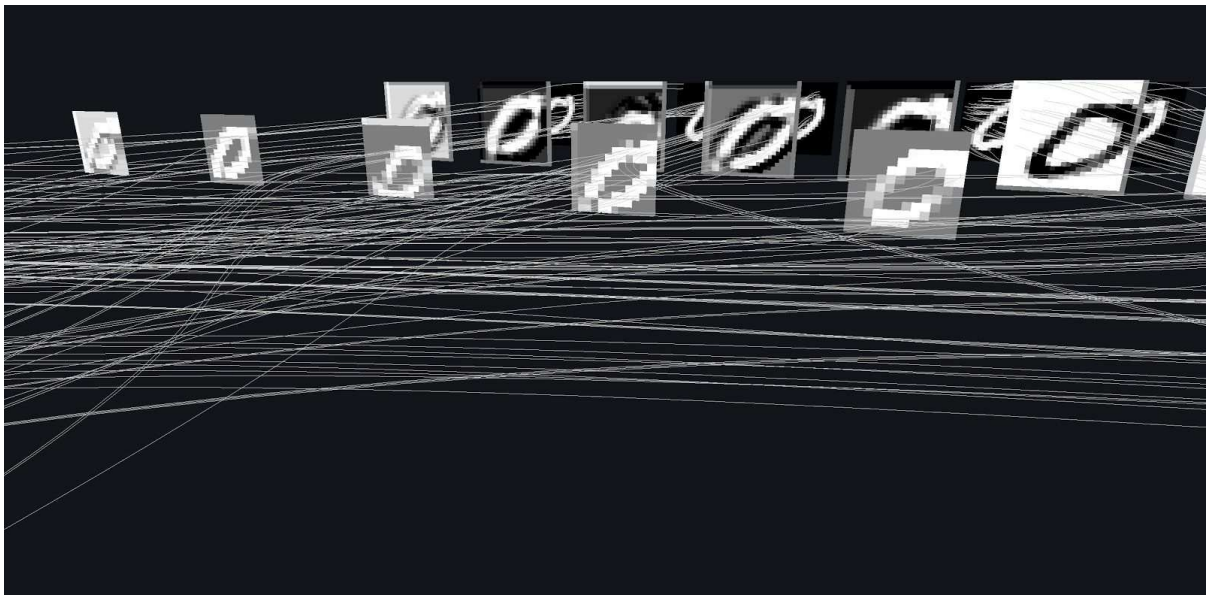

Fig 5.1.6 Close up looking towards inputs.

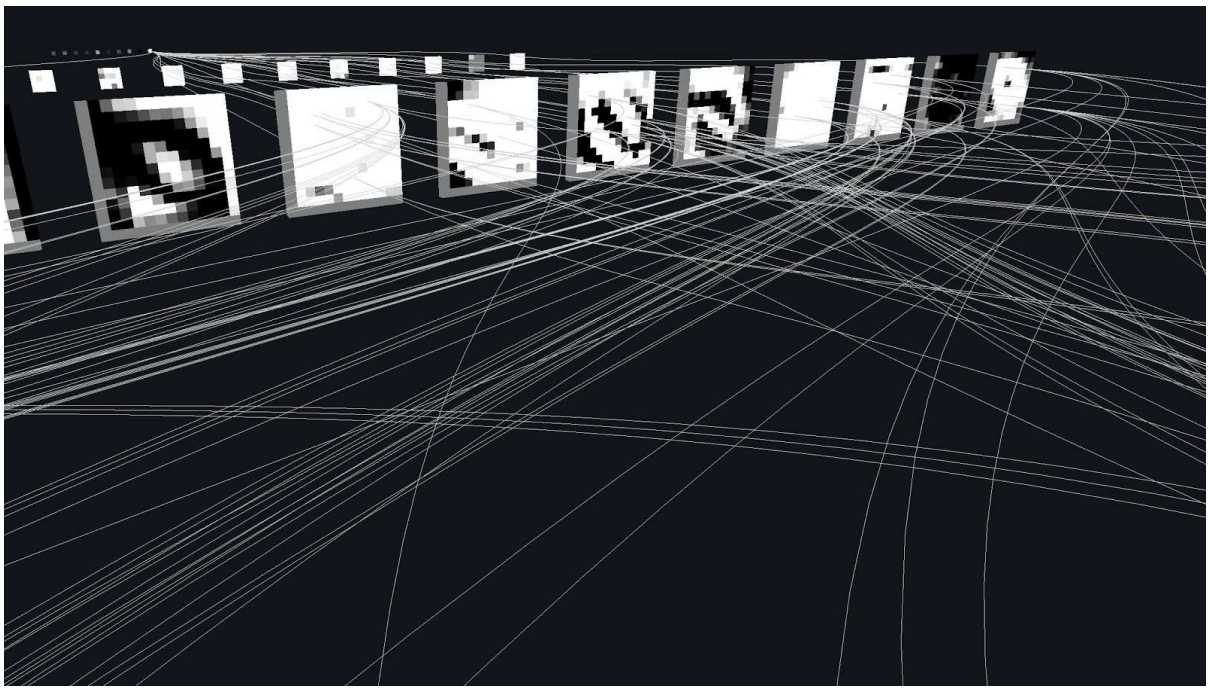

Fig 5.1.7 Close up looking toward top of network.

5.2 Classification of each digit

Examples of classification of each digit except 0 which has been extensively illustrated in section 5.1.

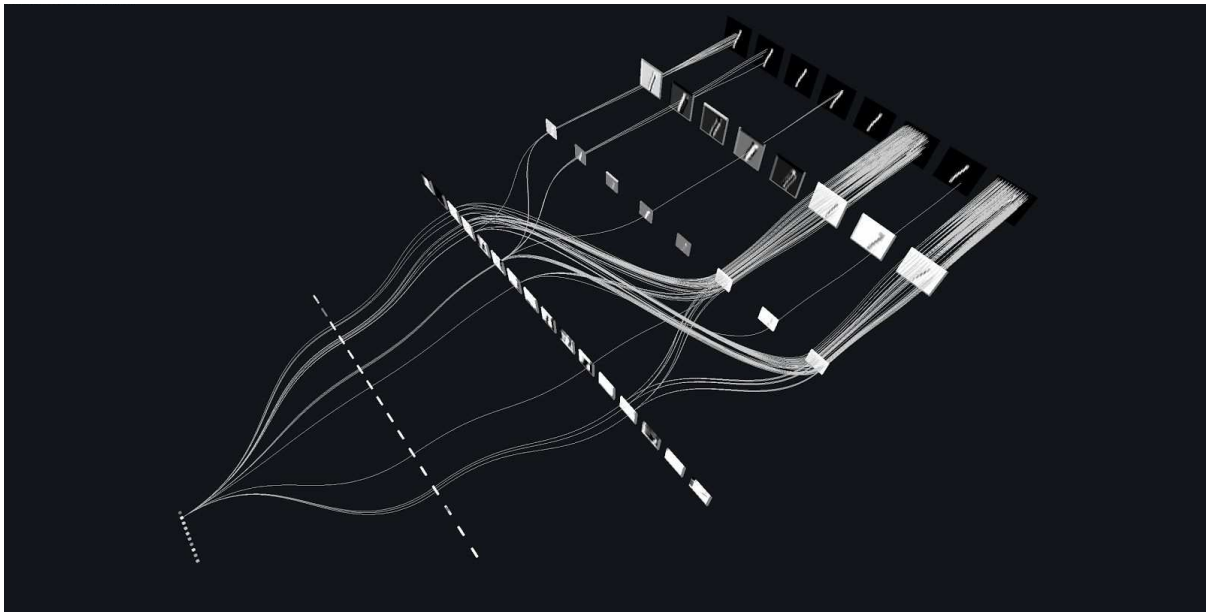

Fig 5.2.1 Classification of the digit 1.

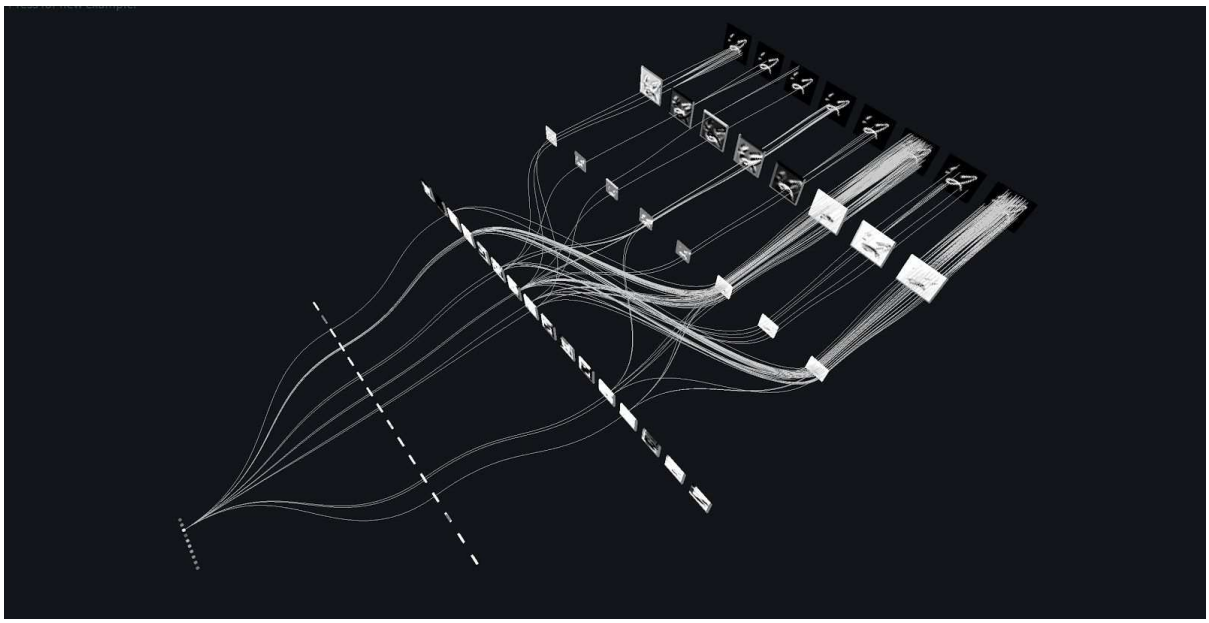

Fig 5.2.2 Classification of the digit 2.

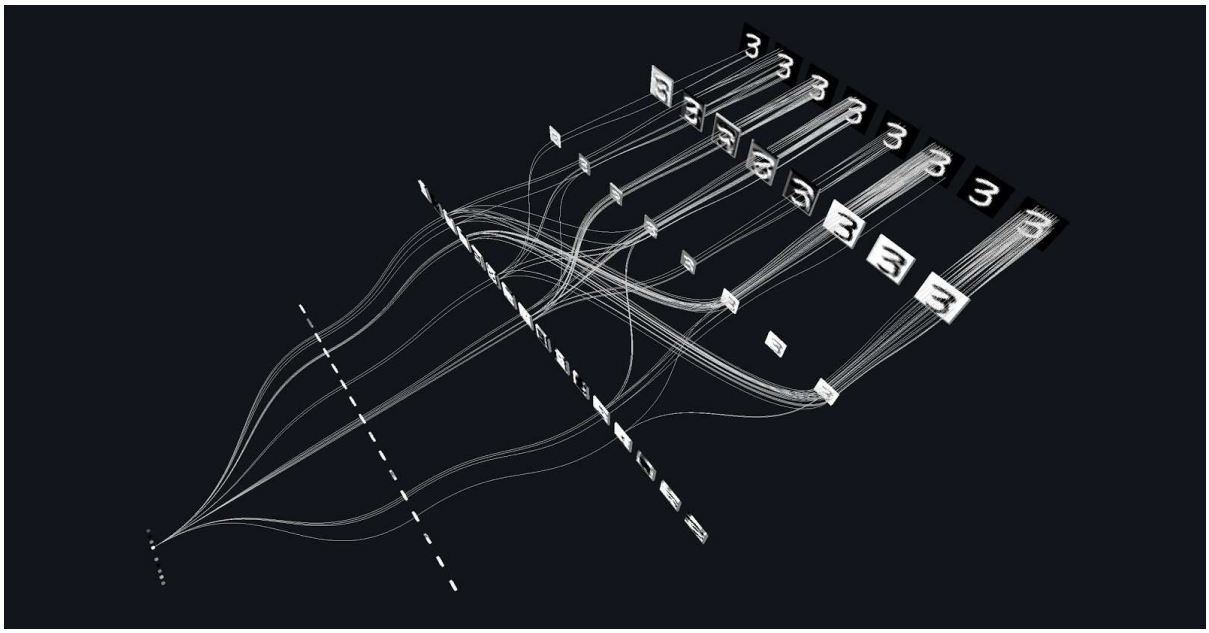

Fig 5.2.2 Classification of the digit 3.

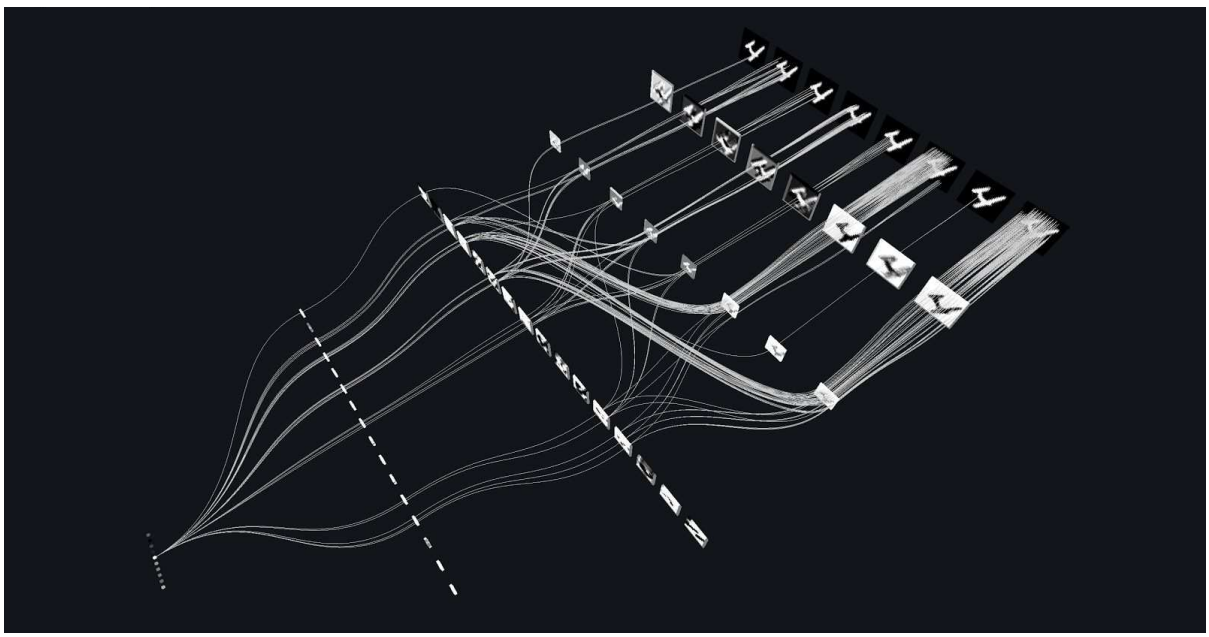

Fig 5.2.2 Classification of the digit 4.

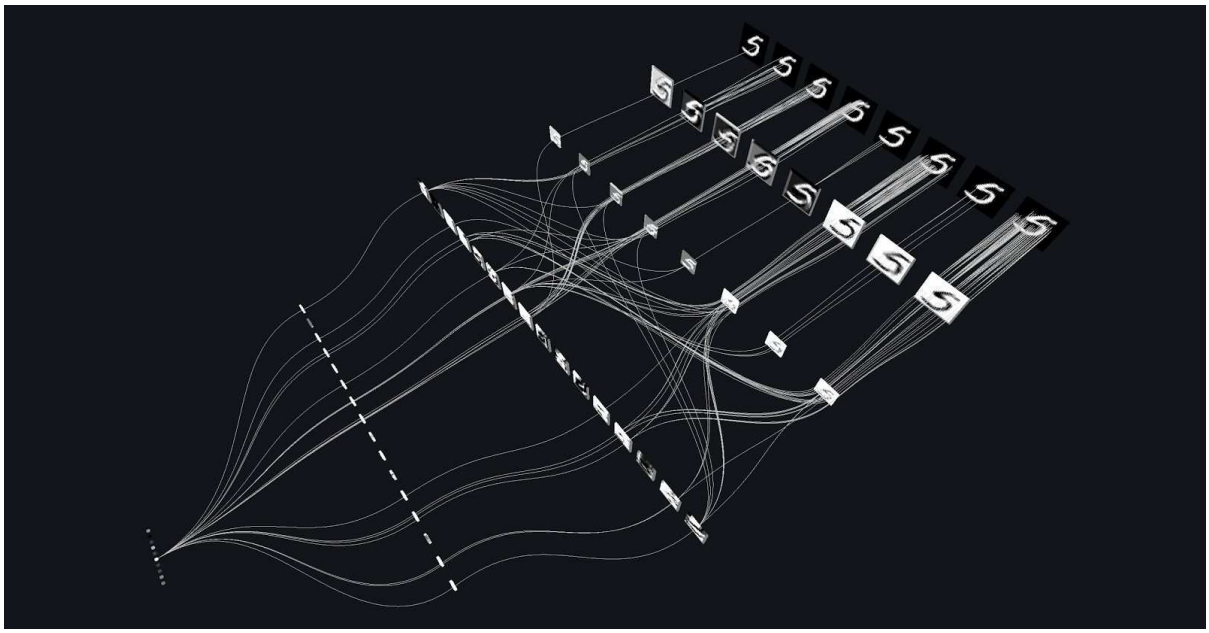

Fig 5.2.2 Classification of the digit 5.

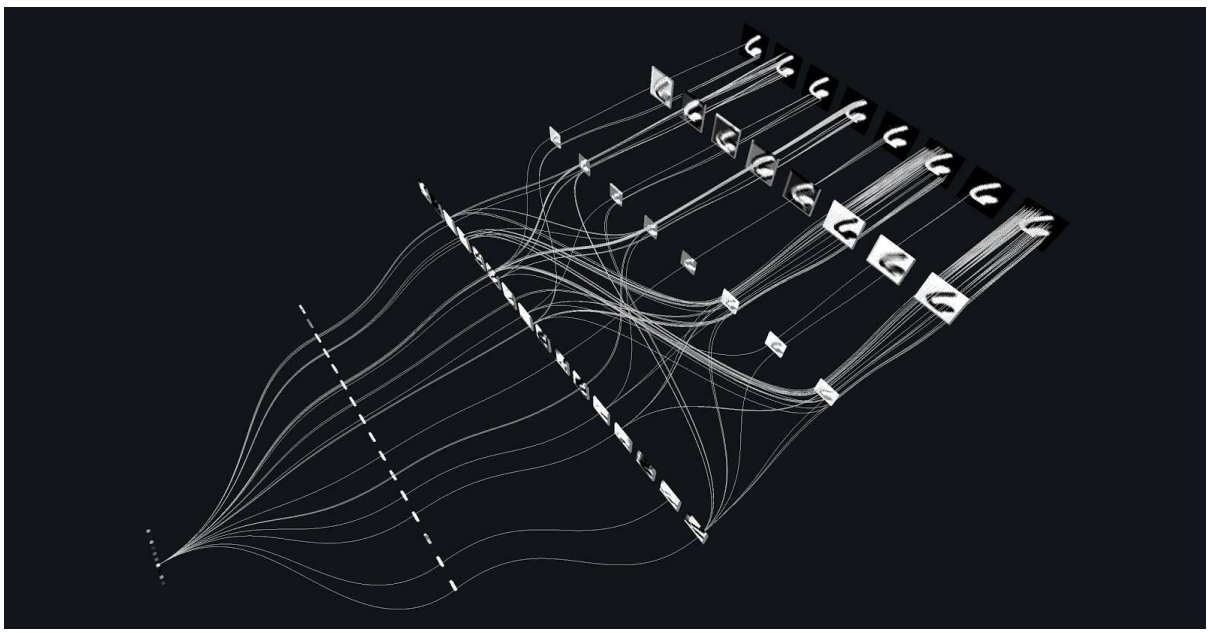

Fig 5.2.2 Classification of the digit 6.

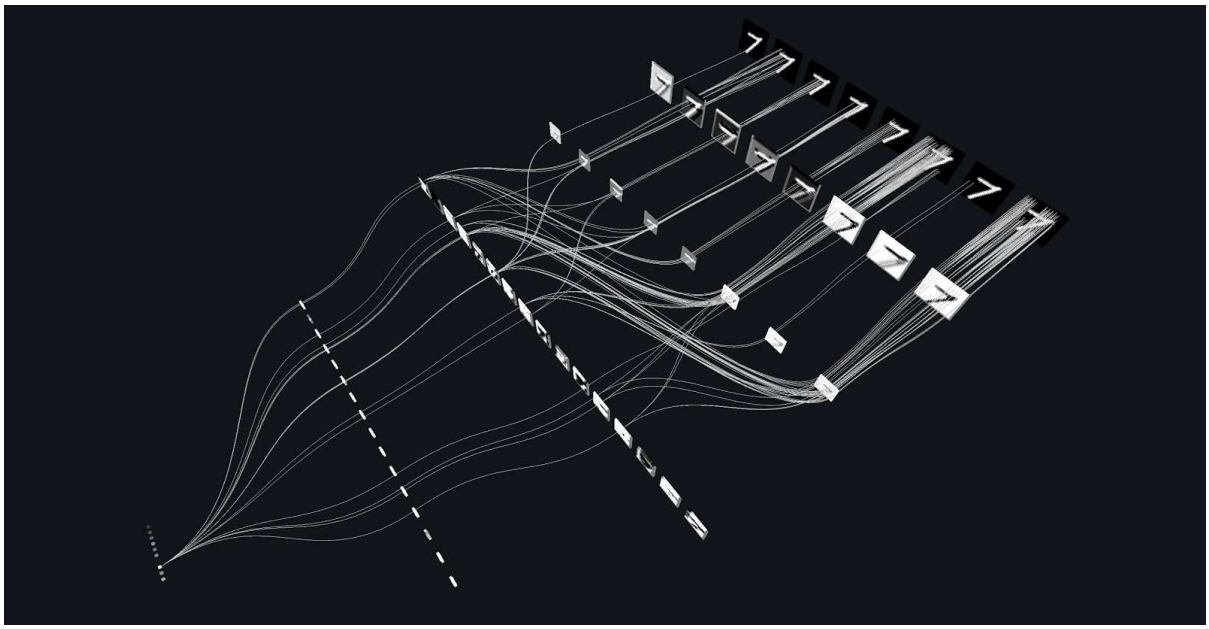

Fig 5.2.2 Classification of the digit 7.

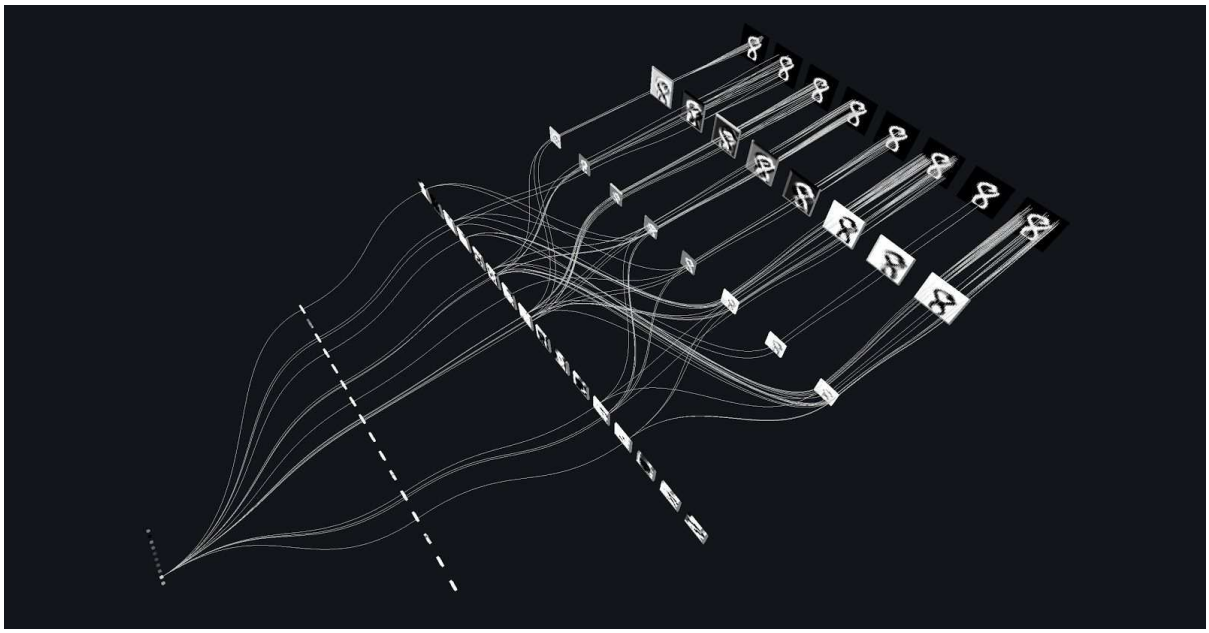

Fig 5.2.2 Classification of the digit 8.

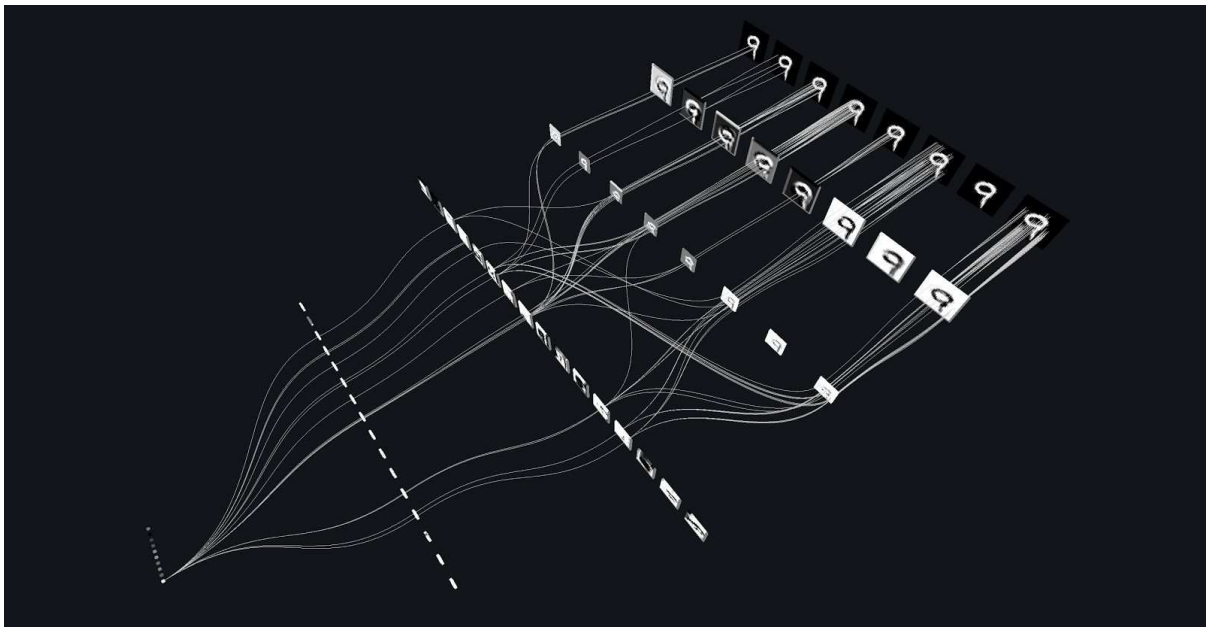

Fig 5.2.2 Classification of the digit 9.

6 Evaluation

In general the visualisation has satisfied at least some of the stated aims, as given in section 3.1 *"to visualise the topological structures that emerge given a data structure that the network has learnt to recognise"*. It is very clear to see in section 5.2 that different topologies of important connections between key nodes in the network emerge given different digits, and it is easy to trace the paths from classification node, through different features, to the pixels in the original input image. No other artificial neural network visualisation does this, the closest to this being Drawnet [Cichy et al. 2016] (see Fig.2.4.6) but this does not show the network topology given an individual input, rather, the average over a large amount of data of different classifications, and examples of features those nodes most respond too.

There are however some shortcomings of the visualisation and its implementation, and possibilities of improvements that could be made in future work, this is covered in the following sections.

6.1 The Visualisation

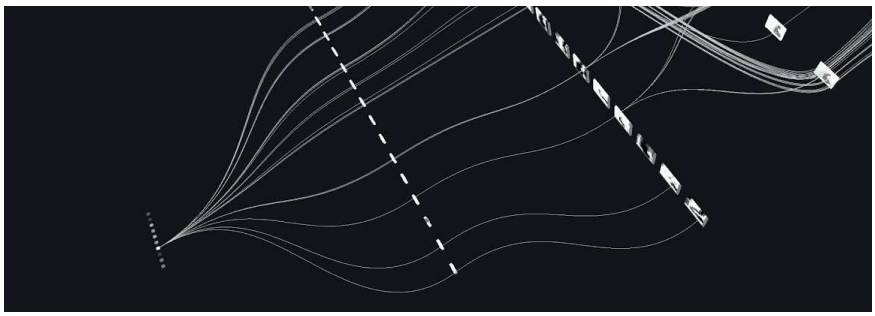

Fig 6.1.1 Branches from the activation tree that stop midway through the network.

One bug that occurs in the visualisation is that occasionally branches will get midway through the network and then abruptly stop. This is because sometimes nodes will not have any going connections going into them above the given threshold. In an attempt to rectify this, in the `treeSearchConv()` function an additional parameter called `gamma` is given, that is used to weight the threshold and is recursively made smaller until a connection is finally drawn. Unfortunately without a restriction on the number of times this could be done, it was easy to exceed the javascript stack depth. In one experiment, if after a certain number of times the threshold is reduced and still no connections meets the threshold, all inputs to that node were drawn. This approach however significantly overrepresented many connections to the network that were not actually important for the classification, and were thus giving a misleading representation of the network topology. Therefore

in the final implementation some connections are allowed to end abruptly. The other alternative was not draw branches that were going to end abruptly, but this would have required a complete restructuring of the depth-first tree search algorithm used in the implementation so it was decided to leave this problem as it only occurs on a couple of nodes, given a couple of digit classifications.

One important aspect of information not visualised is the values that travel down each connection, these are important characteristics of sankey and alluvial diagrams which initially were going to be utilised in the visualisation. Experiments with using these values to adjust line colour and width were performed but the results were not satisfactory.

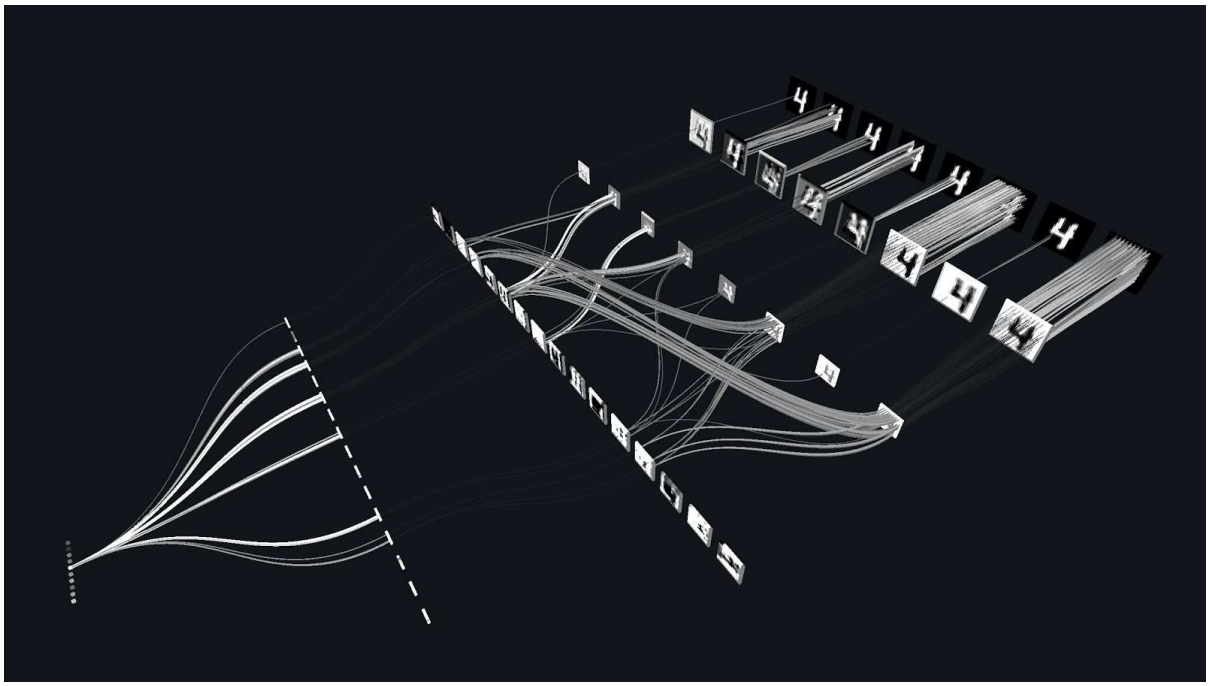

Fig 6.1.2 Line width and colour being rendered according to connection strength.

As can be see in Fig 6.1.2 the problem with simply using the value of each connection to adjust line width and colour is that there are jarring disparities between the strength of the connections in each layer. What this approach does not take into account - that makes sankey and alluvial diagrams so informative - is the value of each connection in the context of the overall energy in a system from layer to layer. This could be rectified as the strength of each connections into a node being in proportion to the strength of the connection from the previous layer, or better still, in proportion to the entire energy of the connection in each layer. This, however, would require an extensive search through every connection in the network to normalise the values prior to the depth-first tree search, which for the sake of in-browser efficiency, was not pursued. The other problem with that approach is that since only connections above a certain threshold are being rendered, adjusting line width or

colour as a proportion of the connections being rendered would be a somewhat misleading representation of what was actually going on in the network. Perhaps rendering all of the connections and using opacity to represent proportional connection strength might work, but unfortunately opacity is not built into three.js so this was not pursued (it can be achieved with custom shaders), and attempting to render every connection in the network would almostly certainly crash most people's web browsers. A method for prerendering the visualisation would be required to take this approach.

6.2 Code Versatility and Reusability

When writing functions for storing values and rendering the visualisation, attempts were made to make them general purpose (not constrained solely to one network architecture) so that they could be used to render other network architectures that can be constructed using ConvNetJS. To an extent this has been achieved, but could be better.

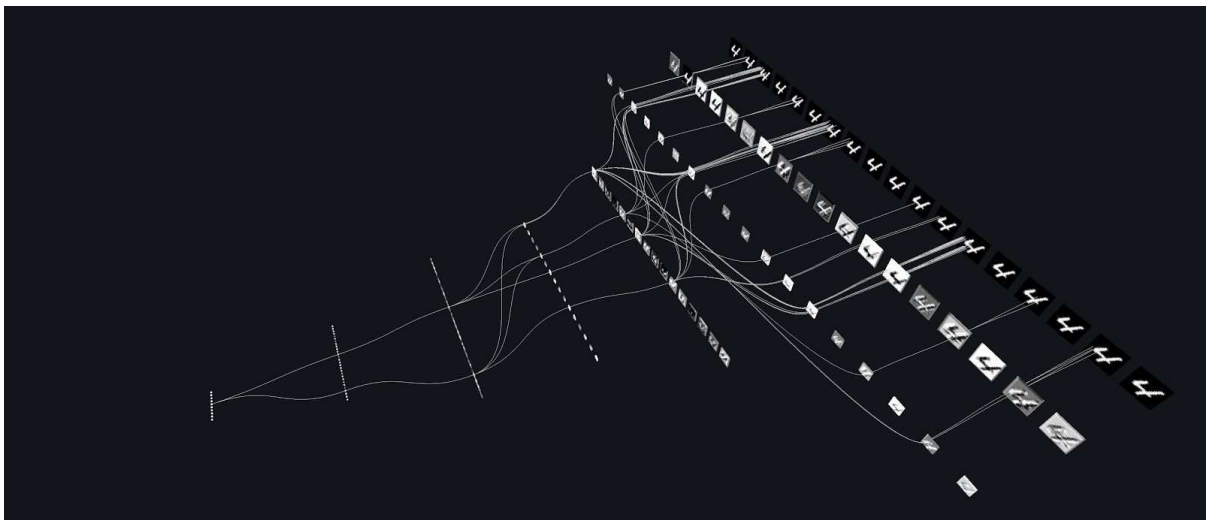

Fig 6.2.1 A network with a deeper architecture, and many more filters (18) in the first convolutional layer.

Experiments with different architectures were performed and the code for doing the visualisation was able to handle network architectures with additional Convolutional and MaxPooling layers, as well as much greater filter depths in the Convolutional layers (as can be seen in Fig 6.2.1). Unfortunately the assumption made was that in the network architectures, there would only be one fully connected layer at the top of the network that was followed by a Softmax layer. Therefore when an attempt to implement an autoencoder architecture (that has many Fully Connected layers) the network could not be rendered properly. Also only the source code of ConvNetJs to was edited to store the additional values in layers that were used in the given network architecture. Therefore this tool would not currently work

for visualising networks using different activation functions, regression, or dropout layers.

6.3 Interactivity

Initially more interactive features were planned, given that this is a web based visualisation and adding interactive features is relatively trivial in javascript compared to other frameworks and languages. As it turned out however, only two elements of interactivity were included, one being a link to press to see a new input, and interactive camera controls (a feature of three.js). There are a couple of reasons for the lack of interactivity.

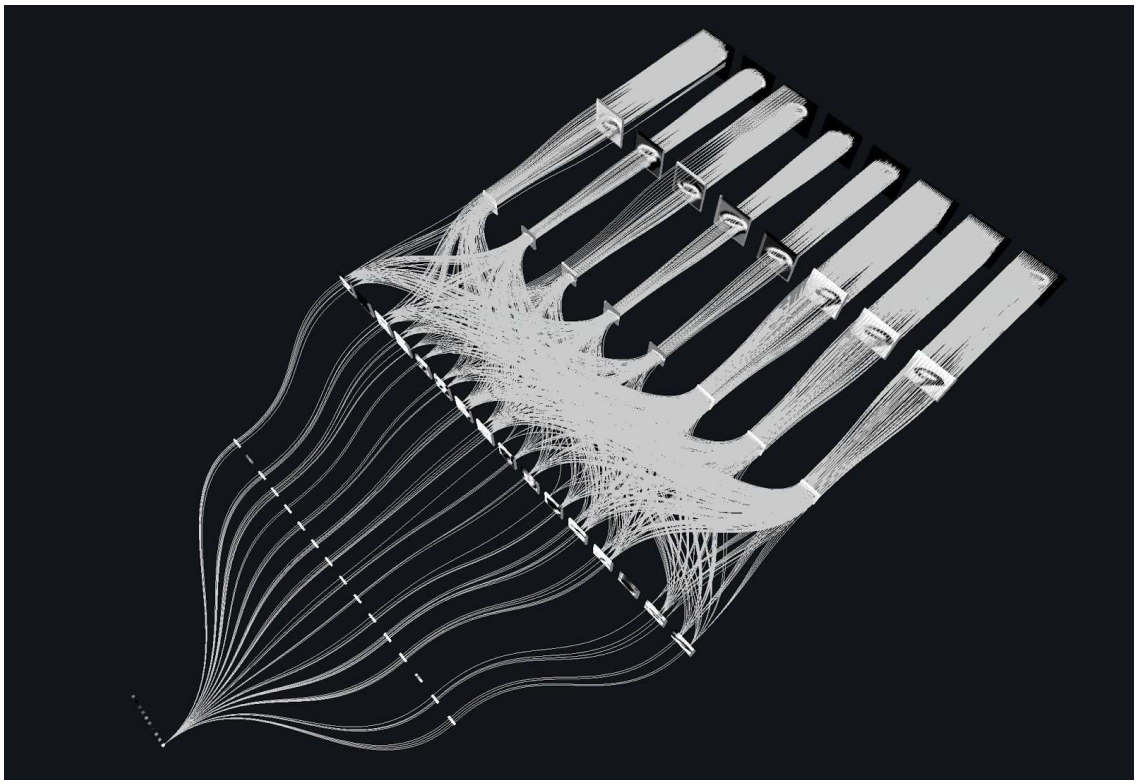

Fig 6.3.1 A rendering of the network with a reduced threshold of 0.5.

One feature intended to be interactive was the threshold for the connection strength. There are three reasons the decision was made not to include this feature in the end. Firstly it is very easy the crash the web browser if the threshold is set too low, and even if it hasn't crashed the web browser it can seriously affect the performance of the rendering and reduce the framerate well into single digits. Secondly, even if it was implemented it would have to be performed either on the next forward pass of the network, or would be so unresponsive (as it deleted all the lines from the scene and recalculated them all) that again it would be sluggish and frustrating from a user experience perspective. Thirdly even if this is not a problem, if the threshold is set too low, so many connections are rendered that it becomes

impossible to differentiate overlapping connections and gain any meaningful insight about the network's topology and how it converges on a classification.

Although the code base is quite versatile and can support some varying network architectures (see section 6.2), having an interactive feature where different network architectures could be defined by the user is not something that could be implemented properly, given the visualisation tools created. Although visualising different architectures is possible, when a network is first initialised the weights are randomly assigned and the network is not capable of frequent, accurate classification until extensive training is carried out. Visualising a network that hasn't been trained is somewhat pointless, as you are not gaining insight into how a network harnesses a topology to make accurate classifications, and training a new network in the browser while rendering the scene is very inefficient and takes a long time, this is why the implementation loads a pre-trained network stored in a JSON file that was trained for an hour when the visualisation was not being rendered.

6.4 Stated Aims That Were Not Achieved

Two aims stated in section 3.1 (albeit secondary, additional aims) were to create an animated visualisation of the process of a neural network learning, and to create a animated visualisation a recurrent neural network as it processes data over time. It would probably not be possible to create an animated visualisation of a convolutional neural network in real time using javascript given the currently processing capabilities of modern browsers and computers. Perhaps it will be possible in the future, but to do it currently, this would need to be pre-rendered and using completely different frameworks and tools. It may be possible to visualise simple recurrent neural networks processing data over time in javascript, but recurrent neural networks are not a feature of ConvNetJS, and a very different approach would be needed to render this; perhaps in 2D, but certainly not by creating many meshes, adding them to a 3D scene and then deleting them and starting over every time the visualisation changes. Thus three.js would probably not be the best tool for creating animated visualisations of recurrent neural networks.

7 Conclusion

On the whole, I am very satisfied with the outcome of this project. The visualisations do indeed expose the various topological structures of activation given different data inputs that I had envisioned but had not seen represented by other artificial neural network visualisations. I like to think that I struck the right balance between an informative visualisation that gives people genuine insight into the implicit logic and 'reasoning' that these networks can be trained to develop, and creating an aesthetically pleasing visualisation. I am proud to be one of the first people to apply some of the more beautiful and advanced techniques used so well in other forms of network visualisation to the task of visualising artificial neural networks. I think it is a potentially very fruitful area of further research, and there is a lot of scope to explore visualisations of much deeper, more complicated and more complicated network architectures that are being explored in the rapidly advancing field of deep learning.

To end on a forward looking note I wanted to include a picture I saw in the news recently, the picture below just won the 2016 Wellcome best scientific image award and is strikingly beautiful, extending both the scientific insight and aesthetic qualities of brain visualisations. It is the latest work from the human connectome project (see section 2.3) and is the best visualisation of a network I have yet come across. Perhaps in the near future there will be a convergence of techniques used to visualise both artificial neural networks and biological neural networks.

Fig 7.1 Wiring the human brain: MRI based tractography to reveal pathways of nerve fibres inside a living person's brain [Anwander 2016]

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. and Ghemawat, S. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems, Software available from http://tensorflow.org

Anwander, A. 2016. Wiring the human brain. http://blog.wellcome.ac.uk/2016/03/16/wellcome-image-awards-2016-winners/ [Accessed 24/03/2016]

Broyez, T. 2013. Sound Sculptures. http://www.visualcomplexity.com/vc/project_details.cfm?id=932&index=932&domain= [Accessed 28/01/2016]

Butler, P. 2010. Visualizing Friendships. https://www.facebook.com/notes/facebook-engineering/visualizing-friendships/469716398919/ [Accessed 27/01/2016]

Carmi, S., Havlin, S., Kirkpatrick, S., Shavitt, Y. and Shir, E., 2007. A model of Internet topology using k-shell decomposition. Proceedings of the National Academy of Sciences, 104(27), pp.11150-11154.

Courville, A. 2015. Undirected Graphical Models. Deep Learning Summer School, Montreal 2015. Available at: http://videolectures.net/deeplearning2015_courville_graphical_models/ [Accessed 28/01/2016]

Clark, J. 2015. Why Deep Learning Will Lead To New, Troublesome Art. https://mappingbabel.wordpress.com/2015/08/30/why-deep-learning-will-lead-to-new-troublesome-art/ [Accessed 28/01/2016]

Di Battista, G., Eades, P., Tamassia, R. and Tollis, I.G. 1994. Algorithms for drawing graphs: an annotated bibliography. Computational Geometry, 4(5), pp.235-282.

Dodig-Crnkovic, G. and Müller, V. 2011. A dialogue concerning two world systems: info-computational vs. mechanistic. Information and computation, 10, p.9789814295482_0006.

Erdos, P., Goodman, A.W. and Pósa, L. 1966. The representation of a graph by set intersections. Canad. J. Math, 18(106-112), p.86.

Fukushima, K. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological cybernetics, 36(4), pp.193-202.

Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H. and Schmidhuber, J. 2009. A novel connectionist system for unconstrained handwriting recognition. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 31(5), pp.855-868.

Graves, A., 2013. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850.

Graves, A., Mohamed, A.R. and Hinton, G. 2013, May. Speech recognition with deep recurrent neural networks. In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on (pp. 6645-6649). IEEE.

Graves, A., Jaitly, N. and Mohamed, A.R. 2013, December. Hybrid speech recognition with deep bidirectional LSTM. In Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on (pp. 273-278). IEEE.

Graves, A. and Freitas, N. D. 2015. Deep learning lecture 13: Alex graves on hallucination with rnns. https://youtu.be/-yX1SYeDHbg?t=49m44s Accessed: 2015-12-10.

Harley, A.W. 2014. Demystifying Deep Convolutional Neural Networks. http://scs.ryerson.ca/~aharley/neural-networks/ [Accessed 28/01/2016]

Harley, A.W. 2015. An Interactive Node-Link Visualization of Convolutional Neural Networks. In Advances in Visual Computing. pp. 867-877. Springer International Publishing.

Harrison, C. 2007. Bible Cross-References. http://www.chrisharrison.net/index.php/Visualizations/BibleViz [Accessed 27/01/2016]

Heart, F., McKenzie, A., McQuillian, J., and Walden, D. 1978. ARPANET Completion Report. Bolt, Beranek and Newman, Burlington, MA.

Heer, J., Bostock, M. and Ogievetsky, V. 2010. A tour through the visualization zoo. Commun. Acm, 53(6), pp.59-67.

Hochreiter, S. and Schmidhuber, J. 1997. Long short-term memory. Neural computation, 9(8), pp.1735-1780.

Hochreiter, S., Bengio, Y., Frasconi, P. and Schmidhuber, J. 2001. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

Honigsbaum, M. 2013. Human Brain Project: Henry Markram plans to spend €1bn building a perfect model of the human brain. The Guardian. 15/10/2013.

Huang, W., Hong, S-H., Eades, P. 2007. "Effects of sociogram drawing conventions and edge crossings in social network visualization", Journal of Graph Algorithms and Applications pp. 397–429

Karpathy, A. 2014. ConvNetJS: Deep learning in your browser. http://cs.stanford.edu/people/karpathy/convnetjs/ [Accessed 29/01/2016]

Karpathy, A. 2015. The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accessed 29/01/2016]

Knuth, Donald E. 2013. “Two thousand years of combinatorics", in Wilson, Robin; Watkins, John J., Combinatorics: Ancient and Modern, Oxford University Press, pp. 7–37.

Koberle, A. 2007. Visualising the Processing Flickr Group. http://www.visualcomplexity.com/vc/project.cfm?id=554 [Accessed 27/01/2016]

Kobourov, Stephen G. 2012. Spring Embedders and Force-Directed Graph Drawing Algorithms, arXiv:1201.3011

Koebe, Paul. 1936. "Kontaktprobleme der Konformen Abbildung", Ber. Sächs. Akad. Wiss. Leipzig, Math.-Phys. Kl. 88: 141–164.

Krizhevsky, A., Sutskever, I. and Hinton, G.E. 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

Kurzweil, R., 1990. The age of intelligent machines (Vol. 579). Cambridge: MIT press.

Kurzweil, R. 2000. The age of spiritual machines: When computers exceed human intelligence. Penguin.

Kurzweil, R., 2005. The singularity is near: When humans transcend biology. Penguin.

Kurzweil, R., 2012. How to create a mind: The secret of human thought revealed. Penguin.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp.2278-2324.

Lee, H., Grosse, R., Ranganath, R. and Ng, A.Y. 2011. Unsupervised learning of hierarchical representations with convolutional deep belief networks. Communications of the ACM, 54(10), pp.95-103.

Lima, M. 2011. Visual Complexity. New York: Princeton Architectural Press.

Lima, M. 2014. The Book of Trees. New York: Princeton Architectural Press.

Markram, H., 2006. The blue brain project. Nature Reviews Neuroscience, 7(2), pp.153-160.

Muller, B. 2006. Poetry on the Road. http://www.esono.com/boris/projects/poetry06/ [Accessed 27/01/2016]

Müller, V. 2012. Computers Can Do Almost Nothing – Except Cognition (Perhaps). Whitehead Lectures on Cognition, Computation & Creativity. Goldsmiths, London. [Attended 24/10/2012]

Noë, A. 2009. Out of our heads: Why you are not your brain, and other lessons from the biology of consciousness. Macmillan.

Ortega, F. 2007. neural network simulator. https://vimeo.com/121008899 [Accessed 28/01/2016]

Riet, H.V.D. 2006. On Bots - Binary Search Tree. http://drunkmenworkhere.org/219 [Accessed 28/01/2016]

Roberto, M. 2010. Spiking Neural Network v1.1(1). https://www.youtube.com/watch?v=lhIdisK7akI [Accessed 28/01/2016]

Saaty, T.L. 1964. The minimum number of intersections in complete graphs. Proceedings of the National Academy of Sciences of the United States of America, 52(3), p.688.

Toga, A.W., Clark, K.A., Thompson, P.M., Shattuck, D.W. and Van Horn, J.D. 2012. Mapping the human connectome. Neurosurgery, 71(1), p.1.

Yildirim, M.D. 2015. Alluvial Diagram + hybrid Modifications / “Sankey Diagram”. http://mezbaha.tumblr.com/post/120106286786/alluvial-diagram-hybrid-modifications-sankey [Accessed 27/01/2016]

Zeiler, M. D. and Fergus, R. 2014. Visualizing and understanding convolutional networks. Computer Vision–ECCV, pp. 818–833.